はじめて学ぶ
Quick BASIC

織田純一著

新紀元社

はじめて学ぶ
Quick BASIC

織田純一著

新紀元社

まえがき

■新しいスタイルのコンピュータ言語

コンピュータ言語の新しいスタイルが生まれました。

Quick BASICです。

BASICというと、「実行が遅い」「構造化ができない」「GOTO文の多用でプログラムが見にくくなる」などの陰口をささやかれてきましたが、ここに紹介するQuick BASICは、全く新しい言語といっていいほど内容を異にしています。

基本的な命令は、互換を取るために従来のBASICに似ていますが、実力は数段上といってもいいでしょう。

「構造化ができない」という点は、今まで邪魔であった行番号を取り払うことにより解消され、無意識のうちに構造化を行なえるようになりました。

また、これにより、プログラムのガンとまで呼ばれたGOTO文を一切使わないでもプログラミングすることができます。

実行速度についても、従来のインタプリタ型のBASICと違い、一行一行実行する度に機械語に翻訳しないため、その速度ははるかに高速です。

■単独で実行できるコンパイル型の言語

Quick BASICでは、どのような方法でプログラムの実行を行なっているのでしょうか。

実は、今流行のC言語などと同じ、コンパイル型の言語になったのです。

つまり、これまで一行一行プログラムを翻訳していたのとは違い、あらかじめまとめて機械語に翻訳しておきます。

その結果、翻訳する時間が短縮でき、ひいてはプログラムの実行速度にも影響を与えるのです。
こうして出来上がったプログラムのもう一つのメリットは、たとえばMS-DOSのFORMATコマンドのように、単独で実行できることです。つまり、従来のMS-DOS版のBASICのように、BASICライブラリが必要ありません。
また、SUBプロシージャやFUNCTIONプロシージャなども、従来のBASICにはなかった概念ですので、十分なページを割きました。

■初めてプログラミングする人のための解説

本書は、こうして生まれ変わったBASICの解説書です。
変数の概念や取り扱い方、条件分岐の記述方法など、Quick BASICの基本を、すべてサンプルプログラム付きで解説しました。
最初は本当に簡単なプログラムばかりですから、そのとおりに自分で入力して、実行してください。だんだんと、プログラミング言語の概念がはっきりしてくるはずです。
最後には、グラフィックの基礎まで扱えるようになっています。

■自分で作るプログラム

最近、パーソナルコンピュータというと市販のパッケージソフトが主流となっていますが、自分のしたいことを100%満たしてくれるソフトはありません。
自分で作ったプログラムなら、画面の色が気に入らなかったり、自分には必要のない機能ばかり目につくなどということはありません。
Quick BASICで自作するプログラムでも、パッケージソフトに負けないくらい、素晴らしい性能を持ったものが作れます。
がんばれば、きっとあなたの満足するプログラムができるはずです。
本書がその手助けになれば、著者としてこれほど嬉しいことはありません。

■簡単なことを積み重ねていこう

最後になりましたが「プログラミングをする」とは、決して難しいことではありません。しかし、最初から複雑なことをすれば、それは難しいに決っています。

一つ一つ簡単なことを積み重ねていくのです。

そうすれば、今までできなかったような難問でも、きっとできるようになります。

1989年3月

織田純一

第1章　Quick BASICとN88BASIC

第2章　Quick BASICのインストール

第3章　Quick BASICを動かしてみよう

第4章　エディタの使い方

第5章　Quick BASICの基礎

第6章　変数の基礎

第7章　Quick BASICの演算

第8章　条件分岐

第9章　繰り返し

第10章　サブルーチン

第11章　高度な変数の利用

第12章　トラッピング

第13章　ファイル操作

第14章　グラフィックス

第1章

Quick BASICと N88 BASIC

従来のBASIC（DISK　BASIC、MS-DOS版N_{88} BASIC）とQuick BASICは、細かな点で微妙に違います。

これから初めてBASICを学ぼうという方にはあまり関係がありませんが、なんらかの形で従来のBASICや他のプログラミング言語を触ったことがある人のために、まず従来のBASICとQuick BASICの違い、すなわちQuick BASICの特長を述べておきます。

1　行番号を意識しなくてもよくなった

従来のBASICでは、行番号を先頭に記述し、その次にステートメントを記述して制御を行なっていましたが、Quick BASICでは、行番号を意識しなくてもプログラミング出来るようになりました。

必要であれば従来どうり行番号を記述する方法でプログラミングも出来ます。

CやPASCALなどのコンパイラ言語では、以前より行番号にとらわれないプログラミングが可能でした。つまり、BASICだけが行番号を付けていたということになります。そこで、BASICでもそれらの言語に準拠するため、行番号を取り払ったということでしょう。

しっかりしたプログラミングを行なえば、行番号など全く必要ありませんが、従来のBASICで作成されたプログラムをQuick BASICに移植する場合には、どうしても行番号が必要になってきます。

そこで、行番号付きでもプログラミングが可能なようになっています。

●従来のBASIC

```
10 'SAVE "CALC.BAS"
20 GOTO 1000
30 CLS
40 PRINT "CALC PROGRAMS"
```

この部分が行番号

●Quick BASIC

```
'Program name : CALC.BAS
CALL TITLE
CLS
PRINT "CALC PROGRAMS"
```

2　サブルーチンに新しい概念の導入

従来のBASICでは、サブルーチンといえば“GOSUB”しかありませんでしたが、Quick BASICでは、この他に“SUBプロシージャ”の概念が導入されました。

GOSUBは、従来どうり行番号とラベルで呼び出しますが、SUBプロシージャの場合では、宣言を行なってサブルーチンを呼び出します。

また、GOSUBの場合には変数をどこで使っても全てのルーチンに影響しますが、SUBプロシージャの場合には、SUBプロシージャ内で宣言した変数はそのSUBプロシージャ内でしか有効でない点が大きく違います。

これらの使い方は、実際にプログラムソースを見ないと分からないと思います。

本書では、第10章で「サブルーチン」について詳しくまとめてあります

●GOSUBを使ったプログラム

```
10 'SAVE "CALC.BAS"
20 GOSUB 1000
30 CLS
40 PRINT "CALC PROGRAMS"
```

```
1000 'TITLE
1010 FOR X = 1 TO 7
1020 COLOR X
```

```
1100 NEXT X
1110 RETURN
```

●SUBプロシージャを使ったプログラム

```
DECLARE SUB TITLE()
'Program name : CALC.BAS
CALL TITLE
CLS
PRINT "CALC PROGRAMS"

SUB TITLE
    FOR X = 1 TO 7
        COLOR X
    NEXT X
END SUB
```

SUVプロシージャ

3　言語自身を拡張できる

従来のBASICは、既存のステートメントや関数を使うことしかできませんでしたが、Quick BASICでは自分で使いたい関数を作成し、他の関数などと同じように使うことができるようになりました。

言い替えれば

言語を拡張できる

ということになります。

方法は、"FUNCTIONプロシージャ"を使って関数を作成し、Quick BASIC付属のライブラリマネージャでデフォルトのライブラリに登録します。

この方法を用いれば「カーソル位置を指定すると同時に画面にメッセージを出力する関数が欲しい」と思えば、LOCATE文とPRINT文を組み合せて関数を作成できます。

「関数を自分で作る」というのは、従来のBASICでは想像も付かないことなので、難しいことのように思えると思いますが、実際に操作を行なうと「な〜んだ、簡単ジャン！」と感じるほど、特別なことは行ないません。

●関数を使ったプログラム

```
DECRARE FUNCTION LP(X AS INTEGER , Y AS INTEGER , Z AS STRING)
                 ¦
                 ¦
DUMMY = LP(10 ,15 ,"ﾊﾟｰｿﾅﾙ DataBase Ver2.10"――――関数呼出し
                 ¦
                 ¦

FUNCTION LP(X AS INTEGER , Y AS INTEGER , Z AS STRING) ┐
    LOCATE X, Y                                         │
                 ¦                                      │ 関数本体
                 ¦                                      │
END FUNCTION                                            ┘
```

4　細かな変更点に注意

Quick BASICでは、従来のBASICで使われているステートメントや関数のほとんどが使えますが、一部、細かな変更で使用不可能となるステートメント・関数があります。

カーソル位置を特定するLOCATE文を例に挙げてみることにしましょう。

従来のBASICでは、LOCATE文のもっとも最小の表現は、“0”から始まりますが、Quick BASICでは“1”から始まります。

LOCATE文の開始位置が違うため、従来のBASICのプログラムをそのまま使用すると、

引数が許される範囲ではありません（ERR=5）

というエラーとなります。

これと同様に、BEEP文などでは、“3”までのオプションがありましたが、Quick BASICでは若干ことなりますので、プログラミングに際し、注意が必要です。

●細かな変更点が加えられたところ

```
190 LOCATE [0,0]―――――― この範囲は許されない
200 PRINT MSG1
210 CLS
230 FOR COUNT = 1 TO 3
240 BEEP [3]―――――― 引数がQuick BASICではサポートされていない
250 NEXT COUNT
        |
        |
```

Quick BASICでプログラミングを行なう際の従来のBASICとの相違点を見てきましたが、このほかにもまだ違う点が少しだけ残っています。

しかし、ここに挙げた相違点を把握しておくだけでもQuick BASICでのプログラミングが楽になると思います。

あとは、問題が生じた時点で解説していきます。

■本書のプログラミングスタイルについて

さて、本書では他言語への配慮も含め、「行番号を意識しないBASIC」ということを念頭において、BASICのプログラミングを解説していきます。

「行番号」があると、ついGOTO文が使いたくなり必要以上にプログラムを複雑化させる恐れがあるため、本書では一切行番号を使用致しません。

ここでもう一つ述べておきます。

GOTO文を頻繁に使ったプログラムにいいプログラムはありません。

GOTO文は無条件にプログラムの制御を変更できるので、プログラムを始めたばかりの方はつい使ってしまう傾向にあるようですが、GOTO文を多用すると、わけのわからないグチャグチャなプログラムになってしまう恐れがあります。

したがって、本書ではGOTO文についても最小限にその使用をとどめることにします。

第2章

Quick BASICのインストール

どんな言語を使うにも、その言語を使えるようにするために、作業ディスクを作成したり、自分のハードウェアの環境にあった設定を行なったりする必要があります。

この作業のことを「インストール」とか「セットアップ」といいます。

ここでは、Quick BASICでプログラミングするために必要な、インストールの方法を解説します。

2・1

必要なハードとソフト

最初に確認しておきたいこと

1　ハードウエアの構成

Quick BASICで、必要なハードウェアの構成は

①コンピュータ本体

ただし、メインメモリ640KB以上でなければなりません。

また、PC-9801シリーズの場合、XA・XL・XL²などのハイレゾリューションモード、LTでは使えません。

②ディスクドライブ

２台か、もしくは１台＋ハードディスク

③高解像度カラー（モノクロ）モニタ

です。

また、

プリンタ
RAMディスク

などがあった方が、効率の良いプログラミング環境が得られます。

2　ソフトウエア

Quick BASICで必要なソフトウェアは

①MS-DOSシステムディスク
②Quick BASICマスターディスク
③ブランクディスク２枚

です。
また、日本語入力を行ないたい場合には、

●ATOK6
●MS-KANJI API仕様に対応しているFEP
VJE-β Ver.2
EgBridgeなど

のいづれかを用意してください。
MS-KANJI API仕様に対応したFEPとは、MS-WINDOWSやOS/2でも使えるように設計されているものです。

2・2

作業ディスクの作成

いつも環境に注意しておこう

1　コンピュータ言語のシステムディスク

どんなコンピュータ言語でもそうですが、プログラミングにとりかかる前に、まず、作業用のシステムディスクを作る必要があります。

コンピュータ言語の場合、プログラムをコンパイルしたりリンクしたり、デバッグするための実行ファイル(.EXEという拡張子のついたファイル）を

BIN

というディレクトリに入れ、また、プログラム開発のためのステートメントや関数などが入っているライブラリファイルを

LIB

というディレクトリに入れる方法が良くとられます。

そして、アプリケーションソフトを使う場合でもそうであるようにシステムディスクを作る場合には

AUTOEXEC.BAT
CONFIG.SYS

を、ソフトの性格と自分の環境に合せて作成しなければなりません。

しかし、Quick BASICでは、この作業をQuick BASICプログラムに入っているセットアップユーティリティ“SETUP.EXE”が行なってくれます。

つまり、“SETUP.EXE”が、作業ディスクにBIN、LIBというディレクトリを作り、そこに必要なファイルを入れ、ルートディレクトリには（ハードディスクの場合はちょっと違いますが）Quick BASICを使うための環境を構成してくれるAUTOEXEC.BAT、CONFIG.SYSを作ってくれるのです。

2　作業ディスクの作成

それでは、“SETUP.EXE”を使って、自分用のQuick BASIC作業ディスクを作っていきましょう。

ここでは2HDのフロッピーディスクを対象としますが、ハードディスクを使う方は、ハードディスクを使う場合の“SETUP.EXE”のメッセージにしたがってください。また、ハードディスクでのQuick BASICの使い方は次の節（「2・3　HDD、FEPを使う場合の環境設定」）で解説します。

①MS-DOSを起動

MS-DOSのシステムディスクを、Aドライブに入れてMS-DOSを起動して下さい（ハードディスクを使っている方は、ハードディスクを使用する場合の指示に従って下さい）。

```
A>■
```

の状態にしておきます。

②ブランクディスク2枚をフォーマットする

まず、2枚のブランクディスクをフォーマットします。ブランクディスクをBドライブに入れ、MS-DOSのシステムを転送するために/Sオプションをつけてフォーマットします。

```
A>FORMAT B:/S↵
```

/Sでシステムを転送するのを忘れないで下さい。

ここで作成したディスクの1枚は、Bドライブにいれたままにしておき、このディスクをQuick BASICの作業ディスクにします。

③セットアップユーティリティの起動

AドライブのMS-DOSシステムディスクとQuick BASICプログラムディスクを交換して、セットアップユーティリティを起動するために

A＞SETUP↵

と入力します。次のような画面になり、セットアップについての注意事項がひととおり解説されます。確認しておいて下さい。

●セットアップの注意事項解説画面1

```
QuickBASIC  セットアップ  ユーティリティ        Ver1.00

----準備----
                                                        (1/3)
◎フロッピーディスクへセットアップする場合、フォーマット済みのディスクを用意し
  てください。

 [フロッピーベースでQuickBASICを利用する方へ]
  あらかじめフォーマット済みのディスクが必要です。
  メディアのタイプによって必要なディスクの枚数が異なります。
  ワークディスクには FORMAT /S スイッチでMS－DOSシステムを転送しておいて
  下さい。システムが転送されていない場合はセットアップはおこなわれません。
  ・HDタイプ(3.5 2HD, 5 2HD)の場合
       1枚：ワークディスク
  ・DDタイプ(3.5 2DD, 5 2DD)の場合
       2枚：ワークディスク、コンパイラディスク
       (FORMAT /9 で 720Kフォーマットにしておく方が良いでしょう。)
◎フォーマットが済みましたら添付のシール(「ワークディスク」、「コンパイラディ
  スク」)をディスクに貼っておいて下さい。他のシールは使いません。御自由にお使い
  下さい。
フォーマット済みのディスクが用意していなければ、CTRL+C でセットアップを中止して
ディスクのフォーマットを行って下さい。

【何かキーを押して下さい】
```

◉セットアップの注意事項解説画面2

```
ＱｕｉｃｋＢＡＳＩＣ　セットアップ　ユーティリティ　　　Ｖｅｒ１．００

---サンプルプログラム---                                     (2/3)
セットアップでは、サンプルプログラムを転送しません．必要に応じて転送をおこなって下さい．

---ライブラリについて---
ＱｕｉｃｋＢＡＳＩＣのコンパイラ(BC.EXE)が通常利用するライブラリは、以下のものです．
BRUN42A.EXE   ランタイムモジュール
BRUN42A.LIB   ランタイムライブラリ
BCOM42A.LIB   スタンドアロンライブラリ

これらを代替数値演算ライブラリ(ALT-MATH)と呼びます．これらは、数値演算コプロセッサの有無に関わらず、一定の速度で高精度な計算を実現します．
数値演算コプロセッサを装着している方のため BRUN42E.EXE,BRUN42E.LIB,BCOM42E.LIB(80?87エミュレートライブラリ)を用意しています．これによってより高速な演算を実現できます．
フロッピーベースでご利用になる場合は、セットアップではエミュレートライブラリは転送できません．必要な場合は、マスターディスクから転送して下さい．

【何かキーを押して下さい】
```

◉セットアップの注意事項解説画面3

```
ＱｕｉｃｋＢＡＳＩＣ　セットアップ　ユーティリティ　　　Ｖｅｒ１．００

---フロントプロセッサの組み込み---
                                                             (3/3)
ＱｕｉｃｋＢＡＳＩＣに日本語入力システムを組み込む場合は、お使いのフロントプロセッサのマニュアルを参照して各自おこなってください．
なお、ご利用可能なフロントプロセッサは、ＡＴＯＫ６、ＭＳ－ＫＡＮＪＩ　ＡＰＩのものです．添付の「日本語入力システムのご使用について」を参考にしてください．

---ＣＯＮＦＩＧ．ＳＹＳ、ＡＵＴＯＥＸＥＣ．ＢＡＴ---

このセットアッププログラムは、フロッピーへセットアップしたとき「ワークディスク」に CONFIG.SYS , AUTOEXEC.BAT を作成します．ハードディスクの場合は、指定されたディレクトリに、NEW-CONF.SYS, NEW-PATH.BAT を作成しますので、これらのファイルを参考にして CONFIG.SYS AUTOEXEC.BAT を修正してください．環境変数については、README.DOC をご参照下さい．

なお、README.DOC には補足の情報や、訂正資料が記載されています．必ずお読み頂くようお願いします．

【何かキーを押して下さい】
```

④セットアップの内容を設定

前の画面で↵キー（↵キーでなくてもかまいませんがSTOPキーはダメ）を押すと、次のような画面になります。

◉セットアップ開始確認画面

```
ＱｕｉｃｋＢＡＳＩＣ　セットアップ　ユーティリティ　　　　Ｖｅｒ１．００
●セットアップを開始しますか［Y/N］？　Y




-----------------------------------------------------------------------------
これより、ＱｕｉｃｋＢＡＳＩＣのセットアップを開始します。
もし、セットアップの途中でＣＴＲＬ＋ＣもしくはＳＴＯＰキーを押した
場合、セットアップ作業を中断します。
その場合は、もう一度 SETUP コマンドを実行して、最初から設定をやり直
して下さい。また、セットアップが終了した後に設定を変更したい場合も、
もう一度 SETUP コマンドを実行し最初から設定をやり直して下さい。
```

ここで、↵キーを押します。

◉環境設定確認画面

```
ＱｕｉｃｋＢＡＳＩＣ　セットアップ　ユーティリティ　　　　Ｖｅｒ１．００
●ハードディスクにセットアップしますか［Y/N］？　N
●ＱｕｉｃｋＢＡＳＩＣをどのドライブにインストールしますか？　B
●マウスを使用しますか［Y/N］？　Y
●マウスの種類を選択して下さい［1:ﾊﾞｽ/2:ｼﾘｱﾙ］？　1
●マウスドライバを選択して下さい［1:MOUSE.SYS/2:MOUSE.COM］？　2


上記の設定でよろしいですか　［Y/N］？Y
-----------------------------------------------------------------------------
マウスをサポートするデバイスドライバの種類を選択して下さい。
メモリ内に常駐するタイプが１（ＭＯＵＳＥ．ＳＹＳ）で、必要に応じて組込み
が可能なタイプが２（ＭＯＵＳＥ．ＣＯＭ）です。
ＭＯＵＳＥ．ＣＯＭは、ＭＯＵＳＥ　ＯＦＦ　でメモリ解放ができる為、ＣＯＭ
形式を使うことをお勧めします。
```

ここでは、すべての問いにそのまま⏎キーを押します。

ただし、ハードディスクにインストールする人は、最初の問にYを入力してください。

⑤作業ディスクの作成

次のような画面になります。

●セットアップ開始画面

```
上記の設定でよろしいですか　[Y/N] ? Y
--------------------------------------------------------------------------
ドライブ B にワークディスクをセットして下さい。
【何かキーを押して下さい】

作業中です。　しばらくお待ち下さい。


A:¥QB_ENV¥QB.EXE --->B:¥BIN¥QB.EXE
```

OKならば、⏎キーを押すと、画面下に、コピーされていくファイルが表示されます。

プログラムディスクからのコピーが終ると、次のような画面になります。

●ライブラリディスクのセット

```
上記の設定でよろしいですか　[Y/N] ? Y
--------------------------------------------------------------------------
ドライブ A にライブラリディスクをセットして下さい。
【何かキーを押して下さい】

作業中です。　しばらくお待ち下さい。


A:¥LIB_A¥BCOM42A.LIB --->B:¥LIB¥BCOM42A.LIB
```

再度、⏎キーです。

この作業が終了すれば、作業ディスクが完成します。

3 作成したディスクの確認

これで「ワークディスク」が1枚（2DDの場合には2枚）作成されました。

はじめに、フォーマットしたもう1枚のディスクには、本書でこれから作っていくさまざまなファイルを入れていきましょう。

では、作業用のシステムディスクの内容を確認してみましょう。

◉ワークディスクのディレクトリ（2HD版）

```
¥ --------------- IO.SYS
                 ¦- MSDOS.SYS
                 ¦- BIN ------------- QB.EXE
                 ¦                  ¦- QB.HLP
                 ¦                  ¦- QB.INI
                 ¦                  ¦- QB.BI
                 ¦                  ¦- MOUSE.COM
                 ¦                  ¦- BC.EXE
                 ¦                  ¦- LIB.EXE
                 ¦                  ¦- LINK.EXE
                 ¦                  ¦- BRUN42A.EXE
                 ¦
                 ¦- LIB ------------- BCOM42A.LIB
                 ¦                  ¦- BRUN42A.LIB
                 ¦                  ¦- BQLB42.LIB
                 ¦                  ¦- QB.LIB
                 ¦                  ¦- QB.QLB
                 ¦
                 ¦- AUTOEXEC.BAT
                 ¦- COMMAND.COM
                 ¦- CONFIG.SYS
                 ¦- README.DOC
```

4　サンプルプログラム

セットアップユーティリティのメッセージにもあったように、ここで作成された作業用システムディスクには、Quick BASICプログラムディスクに収められたサンプルプログラムは転送されません。

サンプルプログラムを参考にしたい人は、サンプルプログラム用のディスクを作成しましょう。

なお、サンプルプログラムは「ライブラリ&サンプルプログラムディスク」のサブディレクトリ

¥SOURCE

にあります。

◉「ライブラリ&サンプルプログラムディスク」のディレクトリ

```
A>DIR B:¥SOURCE

 ドライブ C: のディスクのボリュームラベルは QB42_NO2
 ディレクトリは B:¥SOURCE ――――サブディレクトリ¥SOURCE

.            <DIR>      88-12-15    4:20
..           <DIR>      88-12-15    4:20
BAR       BAS      6331  88-12-15    4:20
CAL       BAS      5595  88-12-15    4:20
QBL       BAS      6970  89-01-15    4:20

WHEREIS   BAS      5453  88-12-15    4:20
ABSOLUTE  ASM      4520  88-12-15    4:20
INTRPT    ASM     15099  88-12-15    4:20
GCOPY     BAS      1012  89-01-15    4:20
      33 個のファイルがあります.
   818176 バイトが使用可能です.
```

2・3
HDD、FEPを使う場合の環境設定

いつもAUTOEXEC.BATとCONFIG.SYSを確認する

フロッピーディスクだけでQuick BASICを使う人は、さきほど作成した作業用システムディスクを起動すれば、すぐにQuick BASICのプログラミングを始められますが、ハードディスクにインストールした場合は、もう少しの作業が必要となります。

また、日本語を使いたい場合は、日本語FEPを組み込む必要があります。

ここでは、この2点についての注意事項を解説します。

1 ハードディスクを使う

セットアップユーティリティを使って作業ディスクを作成すると、AUTOEXEC.BATとCONFIG.SYSは自動的に作成されますが、ハードディスクを使用する場合には、ここで述べる作業が必要になります。

ハードディスクを使用する場合には、AUTOEXEC.BATとCONFIG.SYSがセットアップコマンドで作成したサブディレクトリ、QBの中に作成されます。

ただし、次のようなファイル名になっています。

NEW-CONF.SYS（CONFIG.SYS）
NEW-PATH.BAT（AUTOEXEC.BAT）

このファイルには、「Quick BASICを動かすための最低限の環境」が書かれています。

したがって、このファイルをもとにして、自分のハードディスクの環境に合わせたCONFIG.SYSとAUTOEXEC.BATを作る必要があります。

NEW-CONF.SYSとNEW-PATH.BATを、自分の環境にあわせた内容に設定しましょう。

この作業を終えたら、NEW-CONF.SYSをCONFIG.SYS、NEW-PATH.BATをAUTOEXEC.BATというファイル名にして、ハードディスクのルートディレクトリへコピーします。次のように入力してください。

```
A>COPY A:¥QB¥NEW-PATH.BAT A:¥AUTOEXEC.BAT↵
A>COPY A:¥QB¥NEW-CONF.SYS A:¥CONFIG.SYS↵
```

もし、うまくいかなければ、もう１度、¥QBの中のNEW-CONF.SYSとNEW-PATH.BATを書き直して、再度コピーします。

2　FEPを使う

FEPを使用する場合には、CONFIG.SYSの内容に気をつけてください。

例えば、ATOK6を使用するのならば、CONFIG.SYSに次のような内容を加えて下さい。

●FEPを使用する場合のCONFIG.SYS

```
DEVICE = ATOK6A.SYS
DEVICE = ATOK6B.SYS
```

しかし、フロッピーディスクだけでQuick BASICを使う場合、作業用システムディスクにFEPの辞書を入れようとしても、全く無理です。このような場合には、辞書ディスクを別に作る必要がありますが、本書では日本語は使いませんでしたので、とりあえず、FEPは組み込まなくても大丈夫です。

第3章

Quick BASICを動かしてみよう

どうですか。インストールは終了しましたか？

ここでは、簡単なプログラムを通して、Quick BASICの操作の基本とプログラミングとは何かを学びます。

Quick BASICは、最近はやりの統合環境型コンパイラです。プログラムを書き、コンパイルし、リンクしてという作業が、エディタの中から実行できます。このエディタが、どのようなものかを把握してください。

まず、Quick BASICでプログラムがどのような方法で実行されるのかを学びましょう。

次に、Quick BASICでの実行形式（インタプリタとEXEモデルを作成するコンパイル）について解説します。

3・1
Quick BASICの起動と終了

どんな画面になるか見てみよう

1 起動

■MS-DOSの起動

Quick BASICを起動するには、まず、先ほど作成したワークディスクをAドライブに入れてリセットキーを押します。画面は、単に

```
A>■
```

の状態になり、必要な環境設定がなされたMS-DOSが起動します。
まだ、Quick BASICそのものは起動しません。

■Quick BASICの起動

コマンドラインから次のように入力します。

```
A>QB⏎
```

もし、いつもA>QB⏎とするのが、面倒ならば、次の内容をAUTOEXEC.BATの最後に書きましょう。

◉変更したAUTOEXEC.BATの内容

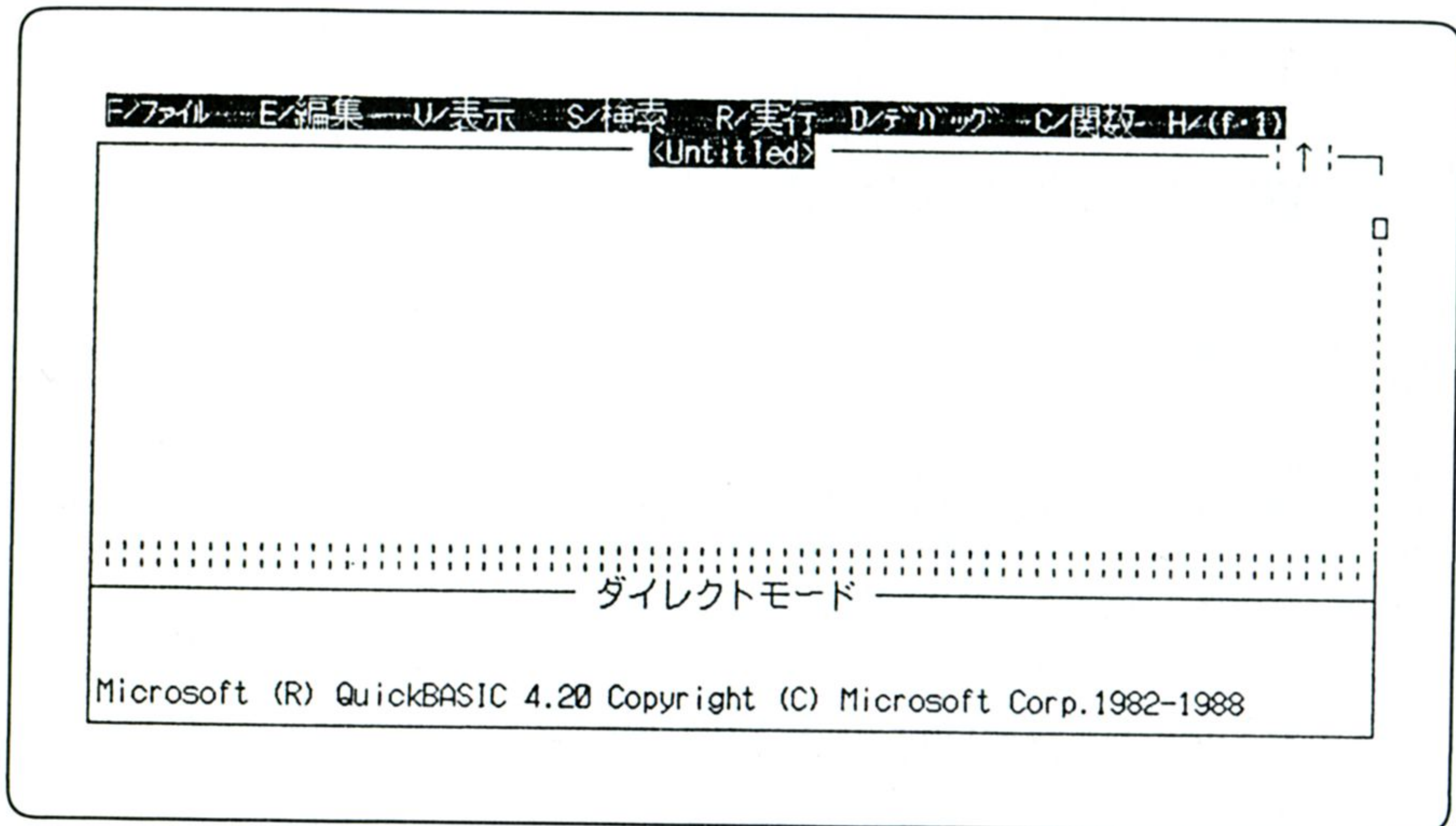
```
SET COMSPEC=A:¥COMMAND.COM
SET PATH=A:¥BIN
SET LIB=A:¥LIB
QB←――――――書き加えた内容
```

これで、MS-DOSの起動に続いて、自動的にQuick BASICが起動するようになります。

■Quick BASICのエディタ画面

Quick BASICが起動すると、次のような画面になります。

●Quick BASICの起動画面

```
F/ファイル  E/編集  V/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f･1)
&lt;Untitled&gt;

ダイレクトモード
Microsoft (R) QuickBASIC 4.20 Copyright (C) Microsoft Corp.1982-1988
```

これが、Quick BASICのエディタ画面です。
基本的には、このエディタの中から

プログラム（ソースリスト）作成
コンパイル
リンク
実行形式のプログラム作成

までを全て行なうことができます（コンパイル・リンクなどは次章で解説）。

2 終了

■終了の方法をキチンと確認しておく

次は、終了の方法を覚えておきましょう。
「起動はできたが終了方法が分からない」といったケースは、良くあるものです。

こうなると、リセットキーを押すしか手がなくなりますし、途中までの作業が、すべて無駄になってしまいます。

Quick BASICを終了するには、

GRPH＋F→X

というキー操作をします。つまり、GRPHキーを押してFキーを押し、Xキーを押します。

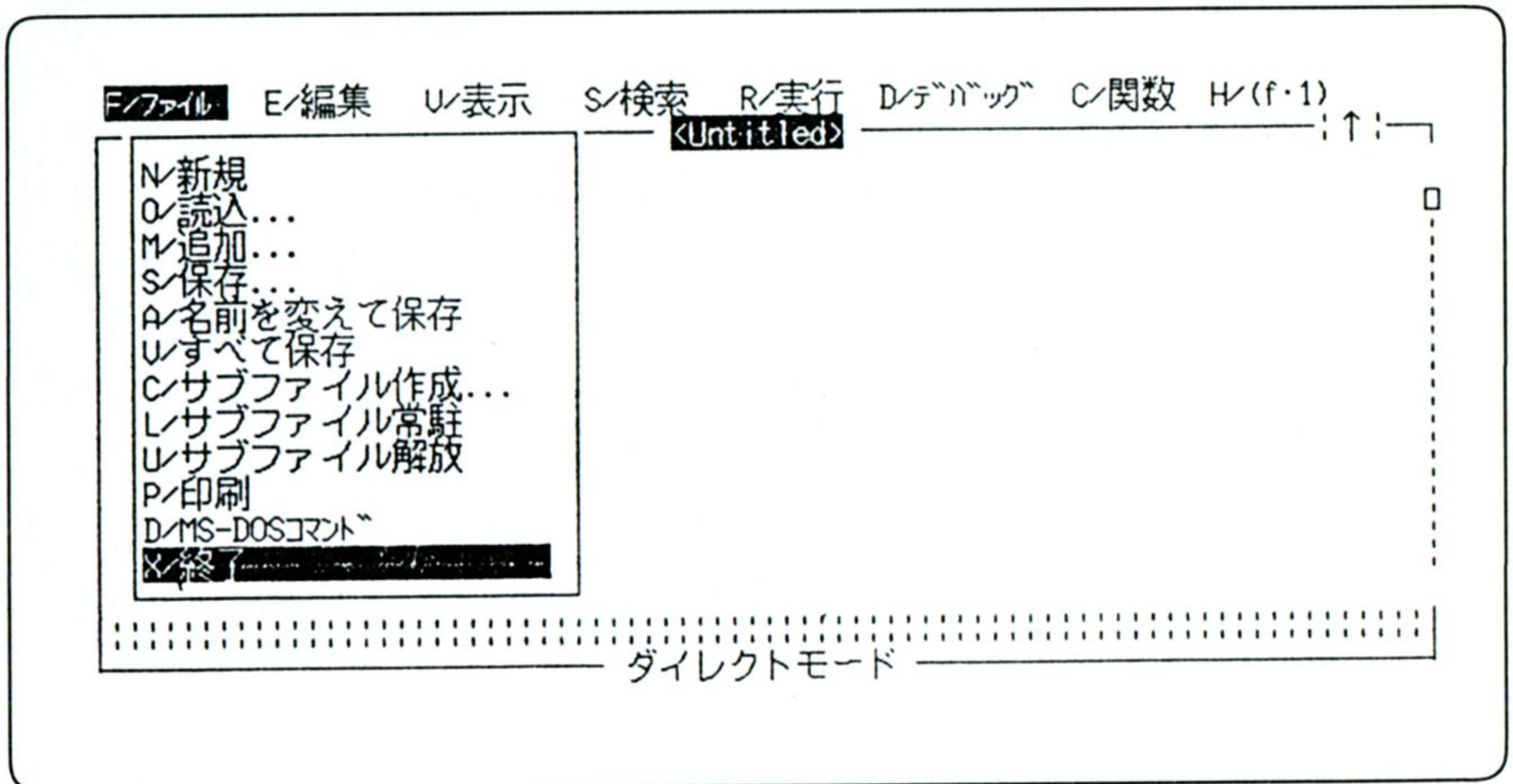
この操作を、画面で確認しておきましょう。

■Quick BASIC終了の操作

①GRPH＋Fキーを押す

次のような画面になります。

●GRPH＋Fメニュー画面

```
F/ファイル  E/編集  V/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f･1)
                              <Untitled>
N/新規
O/読込...
M/追加...
S/保存...
A/名前を変えて保存
V/すべて保存
C/サブファイル作成...
L/サブファイル常駐
U/サブファイル解放
P/印刷
D/MS-DOSコマンド
X/終了
                         ダイレクトモード
```

②X キーを押す

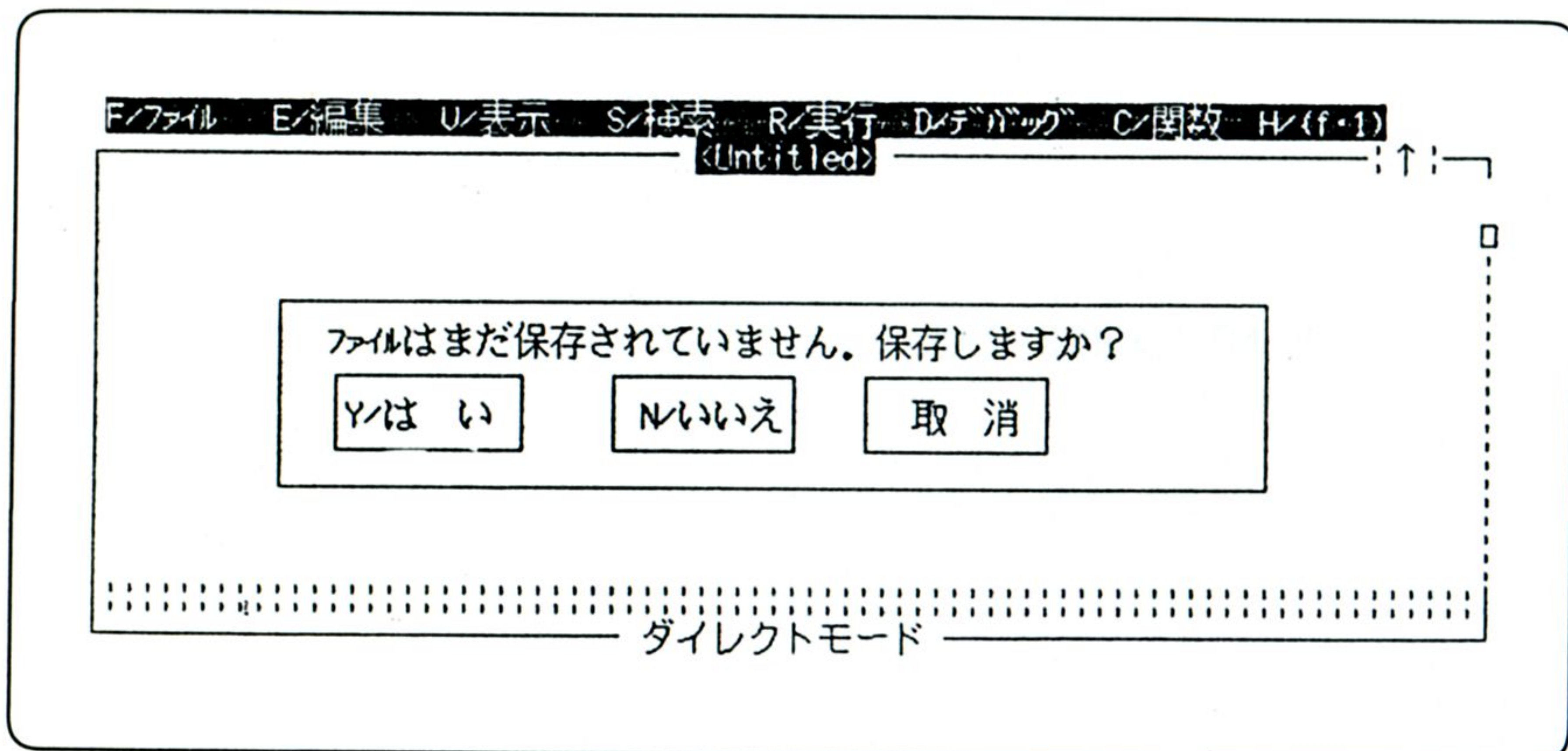
このキー操作で、起動する以前の画面の

A>■

になります。

ここでもし、次のようなメッセージが出力されたら、取り合えず“N”を押してください。

◉ファイル保存確認画面

F/ファイル　E/編集　U/表示　S/検索　R/実行　D/デバッグ　C/関数　H/(f･1)
<Untitled>
ファイルはまだ保存されていません。保存しますか？
Y/はい　N/いいえ　取消
ダイレクトモード

このメッセージは、エディタモード時になんらかのキー入力があったことを示しています。つまり、Quick BASICが「プログラミングが行なわれた」と判断したため、一応プログラムを保存するか聞いてくるのです。

実際にプログラミングを行なったり、プログラムに変更を加えた場合には、必ずそのファイルを保存しましょう。

ただし、ここでは何もプログラミングを行なっていませんので、保存の必要はありません。そこで「保存しない」の意味の“N”を押しました。

3·2

プログラミングを行なう

たった1行でもちゃんとしたプログラム

では、簡単なプログラムを作って、Quick BASICがどのようなものなのかを確認してみましょう。

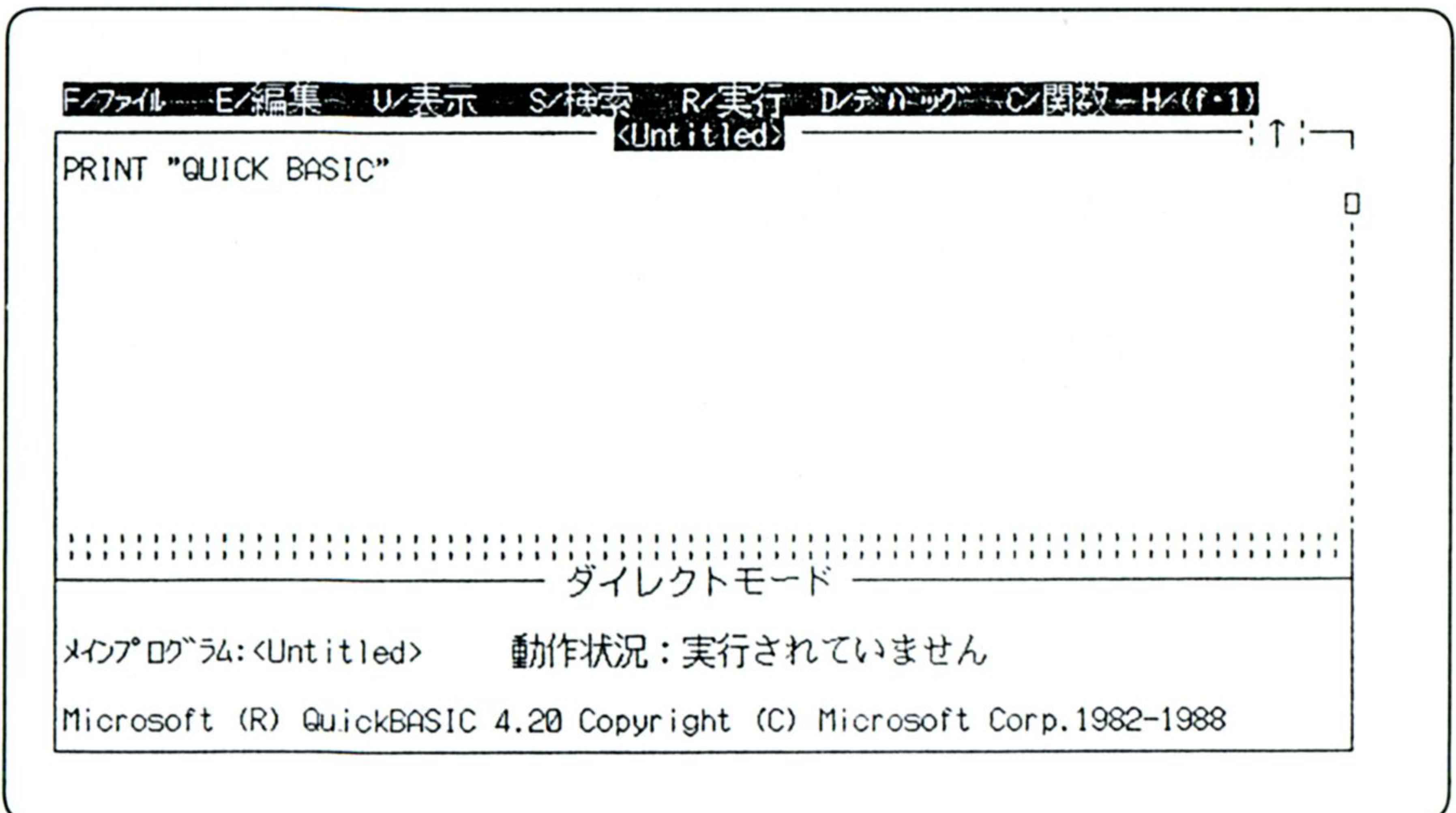
1 ソースリストの入力

Quick BASICを起動します。エディタ画面から

```
PRINT "QUICK BASIC"
```

と入力して下さい。

●ソースリストを入力した状態

```
F/ファイル E/編集 U/表示 S/検索 R/実行 D/デバッグ C/関数 H/(f・1)
&lt;Untitled&gt;
PRINT "QUICK BASIC"

ダイレクトモード
メインプログラム:&lt;Untitled&gt;     動作状況:実行されていません
Microsoft (R) QuickBASIC 4.20 Copyright (C) Microsoft Corp.1982-1988
```

これは、画面に「QUICK BASIC」と表示させるだけのプログラムです。

なおタイプする英字は、かならず**大文字**で行ないましょう。

Quick BASICでは大文字・小文字共に許されます。しかし、ステートメント中に大文字・小文字が入り混じったプログラムは混乱を招きます。

2　ソースリストのセーブ

今入力したソースリストを、ディスクに保存しましょう。

せっかく作ったプログラムを、セーブせずにQuick BASICを終了した場合、そのデータはどこにも残りません。

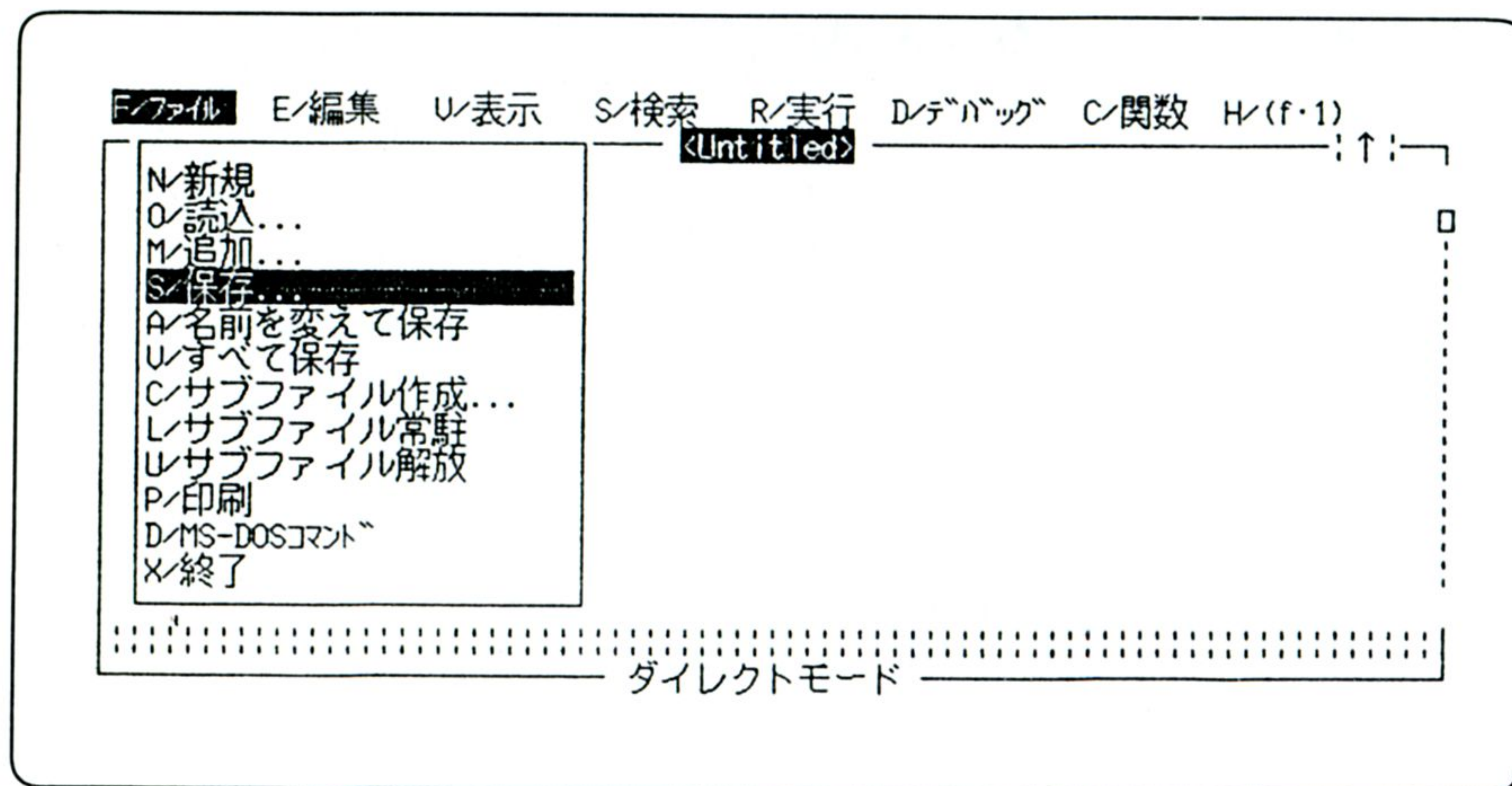
また、暴走、停電などの事故もありえます。

ソースリストを入力したら、ダイレクト実行やコンパイルを行なう前に、必ずディスク上にセーブしましょう。

①GRPH+Fキーを押す

終了の時と同様、GRPH+Fキーを押します。

●GRPH+Fメニュー画面

```
F/ファイル  E/編集  U/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f･1)
                              <Untitled>
N/新規
O/読込...
M/追加...
S/保存...
A/名前を変えて保存
V/すべて保存
C/サブファイル作成...
L/サブファイル常駐
U/サブファイル解放
P/印刷
D/MS-DOSコマンド
X/終了
                          ダイレクトモード
```

②ファイル名を入力

今度は「S/保存」をカーソルキーか、Sキーで選択します。

ファイル名入力ウィンドウが開くので、ファイル名を入力します。ファイル名は

PRG-NO1

と入力して下さい。

◉ファイル名入力ウィンドウ

```
F/ファイル E/編集 V/表示 S/検索 R/実行 D/デバッグ C/関数 H/(f・1)
<Untitled>
PRINT "QUICK BASIC"

    N/ファイル名:  PRG-NO1
    保存形式
    (*) Q/標準ファイル
    ( ) T/テキストファイル
    確 認   取 消

ダイレクトモード
```

本書では以降も、ファイル名はすべて

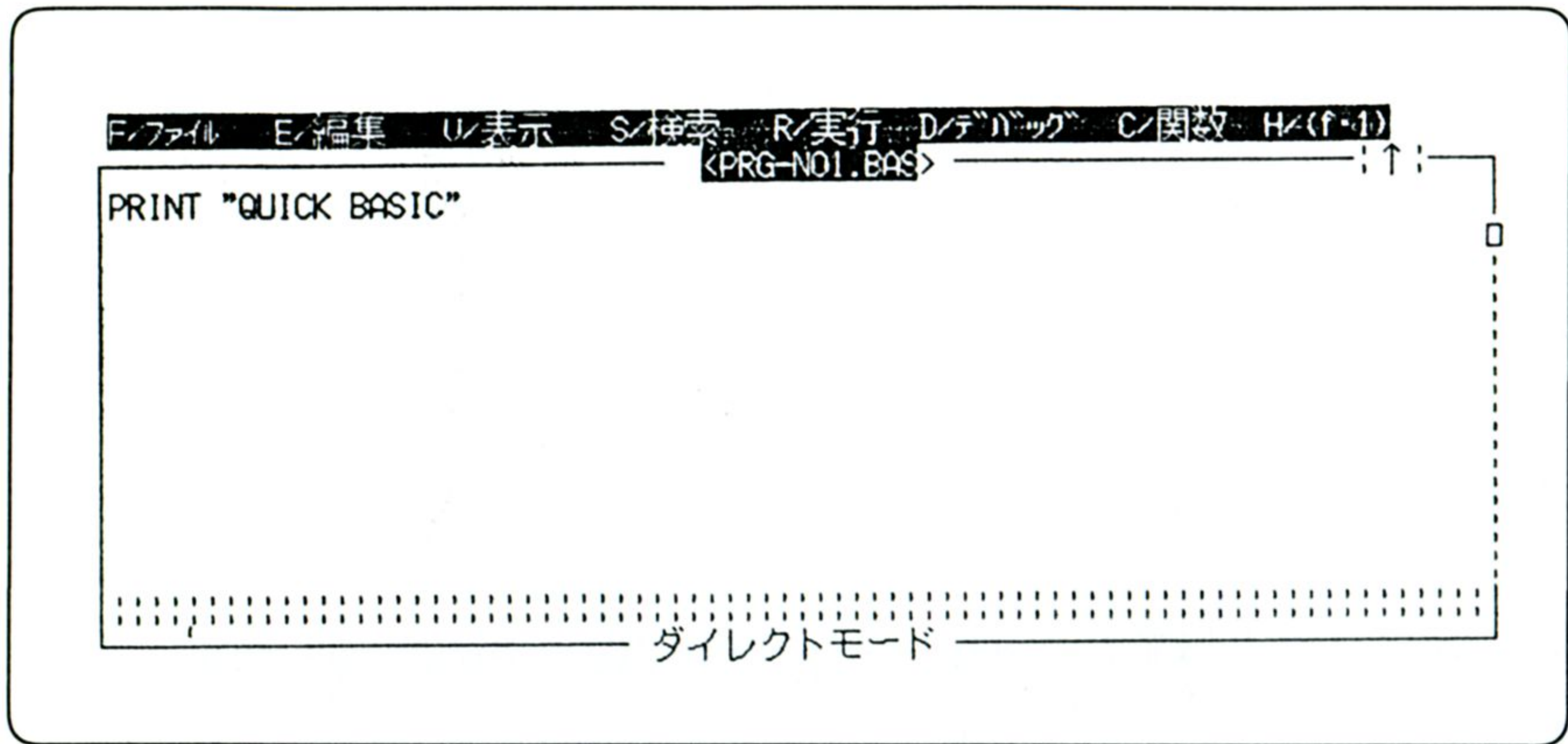
PRG-NO（番号）

という形で進めます。

（ファイル名は、ユーザーが自由に付けることができますので、“PRG-NO”としなくてもかまいません）

ファイル名を入力したら⏎キーを押します。

◉ファイル名の付いた編集画面

```
F/ファイル  E/編集  V/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f・1)
 ──────────────────── <PRG-NO1.BAS> ────────────────────↑
PRINT "QUICK BASIC"

 ──────────────────── ダイレクトモード ────────────────────
```

この操作で、ディスク上に「PRG-NO1.BAS」というファイル名で、今作成したソースリストがセーブされました。

ソースファイル名には、自動的に“*.BAS”という拡張子が付きます。これは、Quick BASICだけでコンパイルできるソースファイルであることを表わします。

3　インタプリタで実行

Quick BASICの、最大の特徴である

インタプリタで作成したプログラムを実行

を行なってみましょう（インタプリタについては次の章で解説）。

f・5キー

を押します。

このコマンドを実行すると画面が一瞬まばたき、次のような画面になります。

◉インタプリタで実行を行なった画面

```
QUICK BASIC




何かキーを押してください
```

確かに

QUICK BASIC

と画面に表示されました。

もし、エラーが表示された場合には、ソースリストに誤りがあるわけです。ソースリストを見直して、再度[f･5]を押してみてください。

ソースリストに手直しを加えた場合には、かならず保存しましょう。

ほんとうに簡単な例でしたが、これでも立派なプログラミングです。これが、プログラミングの基礎の基礎です。

Quick BASICが、どんなものかおぼろげながらも理解していただいたと思います。

3・3

Quick BASICの実行形式

プログラム開発の流れを知ろう

1　コンピュータの言語とは

Quick BASICの実行形式を知る前に、コンピュータが理解できる言葉とは何かについて触れたいと思います。

コンピュータは、人間の書いたプログラムは読めません。コンピュータに理解できるのは、0と1からなる2進数の数字の羅列だけです。

したがって、もっともコンピュータに分かりやすいプログラムを書こうと思うのならば0と1でプログラムを書くべきですが、これを行なうにはよほど数値に強い人でないと、とてもまともなプログラムなど書けないでしょう。

そこで、16進数でプログラムを書くことが考案されました。いったん、16進数でプログラムを書いた後に0と1からなる2進数に変換すればコンピュータにも分かり、人間にも分かりやすいプログラムができます。

しかし、これでも私たちには、まだ不十分です。

16進数で全ての命令を表現し、それを使ってプログラミングするには、やはりある種の天才でなくては無理でしょう。

そこで、誰でもプログラミングできるように「より人間の言語に近い形のプログラム」を「コンピュータが理解できる2進数に変換」するため、様々な言語が開発されました。

これがBASICやC言語、アセンブラなどの「言語」なのです。つまり、本書で学んでいるBASICとは「人間とコンピュータの翻訳者」だったのです。

翻訳者であるQuick BASICは、一種類の翻訳だけをするのではなく、実はいくつかの種類の翻訳方法を知っています。
これが、いくつかの実行形式となるわけです。

2 インタプリタ

先ほど、簡単なプログラムを実行した時の方法です。

[f・5]キー

を押します。
インタプリタは、「同時翻訳」を行ないます。いったん作成したプログラムは、実行直前（1行入力完了時に翻訳する）まで翻訳しません。
したがって、“実行しなさい”の命令があるまで翻訳を行なわないので、翻訳に要する時間がそのまま実行速度に影響します。つまり、「処理が遅い」などの原因となります。
処理が遅くなりますが、インタプリタには次のような利点があります。

①エラーの起こった位置を明確にできる

コンピュータプログラムにはエラーと呼ばれるプログラムミス（バグと呼ばれている）が付きものですが、実行を行ないながら翻訳をするので、エラーとなった時点で実行を中断させることができます。
したがって、プログラムミスを完全になくすまでインタプリタでテスト実行を行ない、プログラムミスがなくなったら、次に解説するコンパイルを行ないます。

②変数の内容の変動を眼で見ることができる

Quick BASICには、ウォッチウィンドウという機能が付いており、実行を行なってからでは追うことのできない、変数の内容を眼で見ながら実行させることができます。

これも、プログラムミスを発見するための手段に使われますが、コンパイルを行なった後では、ウォッチウィンドウを使うことができなくなります。

3　コンパイル

コンパイルは、一括翻訳を行ないます。

インタプリタがプログラムの実行を行なう直前まで翻訳を行なわないのに対し、コンパイルは、翻訳をすでに終らせた状態でプログラムを保存します。したがって、翻訳が終了しているために翻訳にかかる時間が節約できるので、プログラムの処理スピードが向上します。

ただし、インタプリタのように同時翻訳していないために、変数の内容を眼で見たり、またエラーがあってもどこにあるのか分かりません。

よって、実際にプログラミングを行なう場合には、インタプリタで実行テストを繰り返し、完全にバグがなくなったら、コンパイルして一括翻訳を行ない実行させます。

また、Quick BASICのコンパイルには

ランタイム分離型
独立型

という2つの方法があります。

■ランタイム分離型

ライブラリと呼ばれるBASICの基本プログラム部分と各ユーザーが作成したプログラムを分離して保存します。一括翻訳を行ないますが、BASICの基本プログラムが一緒に保存されていないので、実際に実行する場合には、ライブラリがないと動かすことができません。

■独立型

独立型で作成されるプログラムは、先のランタイム分離型で別々に保存されていたBASICの基本プログラムを自分自身に取り込みますので、自分自身だけで実行可能できるメリットがあります。

4　本書のプログラム開発の流れ

ランタイム分離型EXEモデルは、プログラムサイズが小さいため、プログラムのロードには時間がかかりません。独立型EXEモデルは、プログラムサイズが大きくなるためプログラムのロードに、やや時間がかかります。

しかし、C言語などではほとんど独立型の実行形式プログラムを作成しています。

本書でも、独立型の実行形式プログラムを作成します。

すでに、ソースリストの作成から、インタプリタでの実行までは行ないましたが、この流れは次のようになります。

1　ソースリストの作成・保存

GRPH+F → S

2　インタプリタで実行（動作確認）

f･5キーを押して実行

3　コンパイル

ランタイム分離型	GRPH+R→ ↲
独立型	GRPH+R→X→GRPH+A→↲

5　EXEファイルの作成

■EXEファイルを作る

それでは、先程作成したPRG-NO1.BASを実際にコンパイルしてみましょう。

すでに、Quick BASICを一度終了してしまっている場合には、再度Quick BASICを起動し、

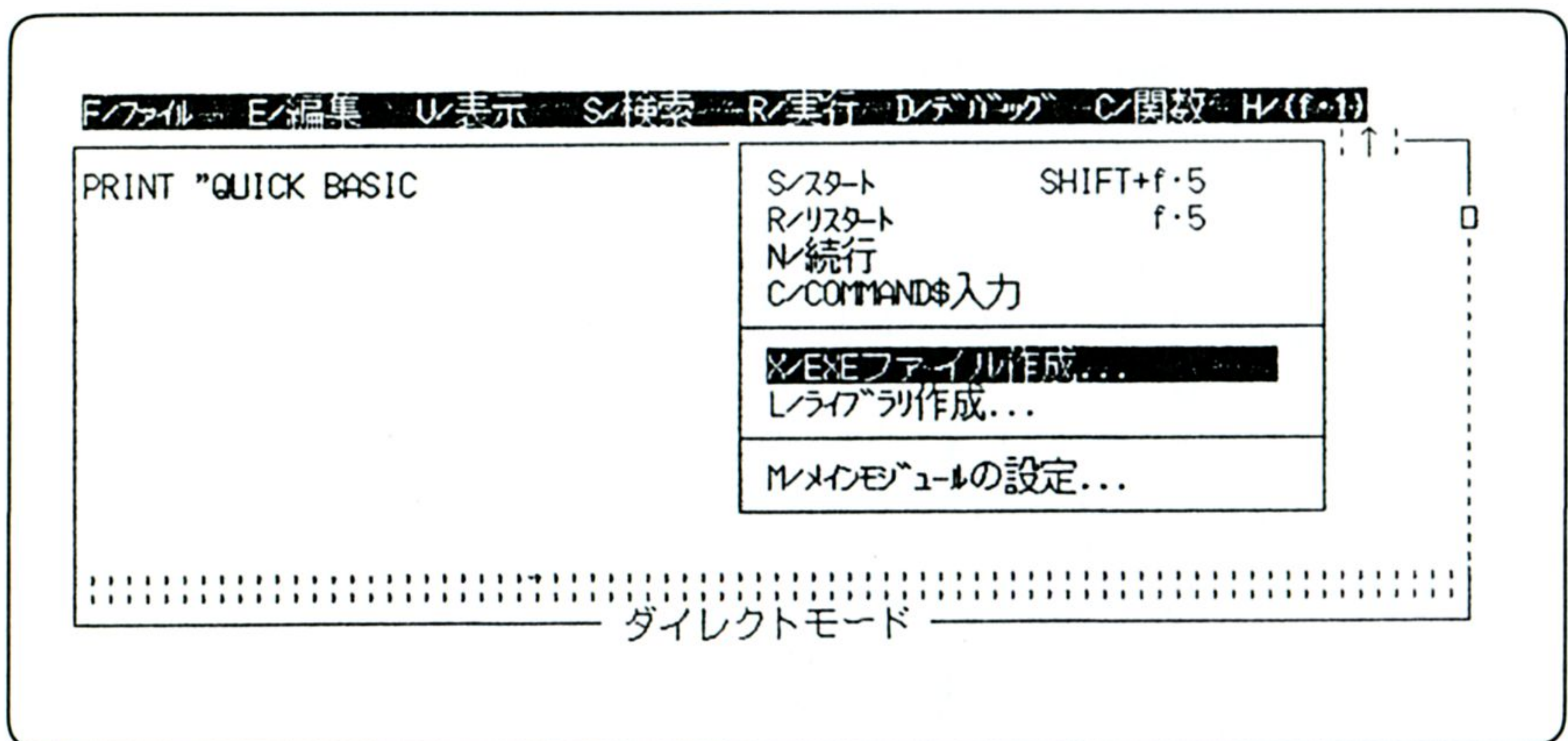
GRPH＋F→O

で、PRG-NO1.BASをエディタ画面に読み込んで下さい。ファイル名を入力すればPRG-NO1.BASを読み込めます。また、TABキーを押すと、カーソルでファイルを選び、⏎キーで読み込めます。

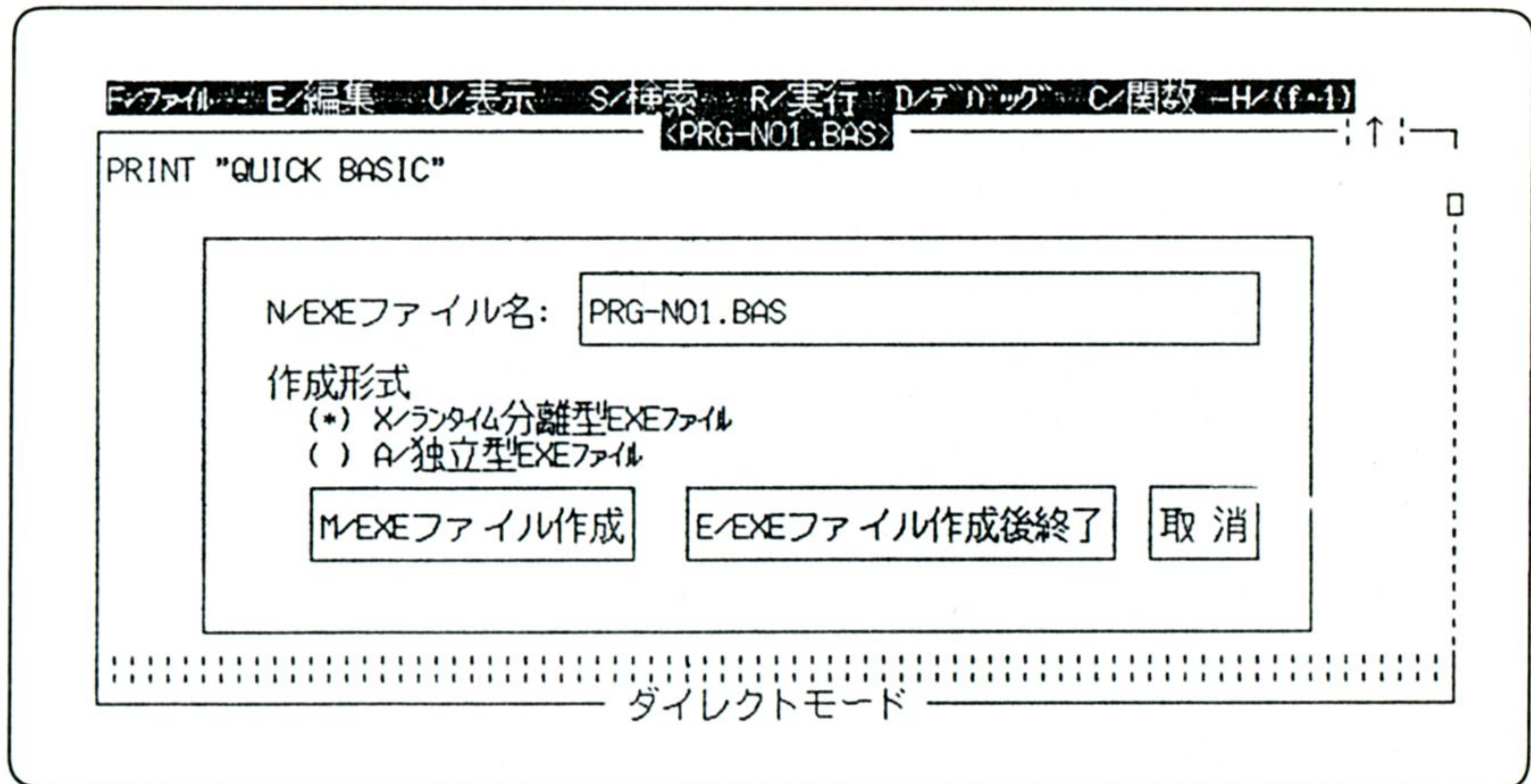
GRPH＋R

というキー操作で、次のような画面になります。

●GRPH＋Rのメニュー画面

```
F/ファイル  E/編集  V/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f・1)
PRINT "QUICK BASIC        S/スタート             SHIFT+f・5
                          R/リスタート                  f・5
                          N/続行
                          C/COMMAND$入力
                          X/EXEファイル作成...
                          L/ライブラリ作成...
                          M/メインモジュールの設定...
                        ダイレクトモード
```

ここで、カーソルで「X/EXEファイル作成...」を選択するか、Xキーを押すと、次の画面になります。

●EXEファイル作成のファイル確認画面

```
F/ファイル  E/編集  U/表示  S/検索  R/実行  D/デバッグ  C/関数  H/(f・1)
                         <PRG-NO1.BAS>
PRINT "QUICK BASIC"

    N/EXEファイル名: PRG-NO1.BAS

    作成形式
     (*) X/ランタイム分離型EXEファイル
     ( ) A/独立型EXEファイル

    M/EXEファイル作成   E/EXEファイル作成後終了   取 消

                         ダイレクトモード
```

ファイル名を確認して（変更したければ新たなファイル名を入力)、ランタイム分離型のEXEファイルを作るのか、独立型のEXEファイルを作るのかを選択します。

この選択は

ランタイム分離型のEXEファイル	**そのまま**↵
独立型EXEファイル	GRPH＋A→↵

で、できます。

ここでは、独立型EXEファイルを作るのでGRPH＋Aです。アステリスクの位置が、独立型の方に移ればOKです。

↵キーを押すと、コンパイル・リンクが実行されます。画面には、その過程が英語で示され、終了すると元のエディタ画面に戻ります。これで、あなたがつくったプログラムである

PRG-NO1.EXE

という実行ファイルが作成されたはずです。確認してみましょう。

■実行可能ファイルのテスト

どんな時にも、まず、ファイル名の確認から始めるのが基本です。ディスクには、

.BASのついたソースリスト（これをコンパイルする）
.OBJのついた中間ファイル（これをリンクする）
.EXEのついた実行ファイル本体（これで実行できる）

が存在するはずです。
では、確認してみましょう。

●PRG-NO1.EXEの確認

```
A>DIR↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND   COM     24931  88-07-13    0:00
BIN              <DIR>   89-03-13   14:11
LIB              <DIR>   89-03-13   14:11
README    DOC      7832  89-01-15    4:20
AUTOEXEC  BAT        61  89-03-13   14:12
CONFIG    SYS        47  89-03-13   14:12
PRG-NO1   BAS       421  89-03-15   15:16 ―― 保存したソースリスト
PRG-NO1   OBJ       661  89-03-15   15:16 ―― EXEモデルを作るまでの中間ファイル
PRG-NO1   EXE     38517  89-03-15   15:16 ―― EXEモデル本体
                                              このファイルだけで実行させることができる
        9 個のファイルがあります.
    73728 バイトが使用可能です.

A>PRG-NO1↵ ―――――― 実行してみる
QUICK BASIC ―――――― 確かに実行された

A>COPY PRG-NO1.EXE B:↵ ―――― 何も入っていないディスクにプログラムをコピー
        1 個のファイルをコピーしました.


A>B:↵ ―――――― Bドライブに移る

B>DIR↵
```

```
 ドライブ B: のディスクのボリュームラベルはありません.
 ディレクトリは B:¥

PRG-NO1 EXE     38517  89-089-03-15   15:16
         1 個のファイルがあります.
  1210368 バイトが使用可能です.

B>PRG-NO1⏎ ――――― "PRG-NO1" だけで実行
QUICK BASIC――――独立型なので自分自身だけで実行できる

B>
```

PRG-NO1.EXEはうまく実行できましたか？

これが、Quick BASICの基本であり、また統合環境版コンパイラの基本です。どうですか、ちっとも難しくなんかありませんね。

本書では、小さなプログラムを実際に作りながら、Quick BASICを学んでいきますが、すべてのプログラムを入力し、実行してみて下さい。すべてのソースの実行ファイルを作る必要はありませんが、.BASのついたソースファイルは全部保存しておきましょう。

どんな大きなプログラムでも、小さなプログラムの集合体です。あなたが大きなプログラムを作るときに利用できる小さなプログラムがあれば、それを再び作る必要はないわけですから。本書でも、似ている部分は、どんどん、すでに作ってあるものから流用して下さい。

さて、Quick BASICのプログラム開発では、これまで見てきたようにエディタが大きな力を発揮します。

ちょっと前までは、ソースの作成、コンパイル、リンクなどを、MS-DOSの真黒な画面からいちいち、手で入力し、わけのわかんないエラーメッセージと仲よく（？）付き合いながらやっていたのです。

次の章で、Quick BASICを使いこなす基本となるエディタについてまとめておきます。

第４章

エディタの使い方

ここでは、Quick BASICのエディタコマンドについて解説します。

エディタについて、気になった時、ひととおり読んで見てください。今は、読み飛ばしても結構です。

Quick BASICのみならず、統合環境型のコンパイラでは、プログラミングを行なうにしても、コンパイルをするにしろ、またデバッグをするにしろエディタを使いこなすことが基本となります。

エディタに関するむずかしい問題は、いまのところ分らなければ、すべて避けて通りましょう。本書で、Quick BASICのプログラミングの基礎を学ぶ分には、ソースを入力したら、それをセーブして、f・5で実行するだけでOKです。

初めから、豊富な機能を自由に使いこなすのは無理でしょうが「こんなことはどうしたらできるだろう」「このメニューはどういう意味だろう」などという時には、このリファレンスを利用して下さい。

もちろん、チャレンジ精神旺盛な方は、便利な機能を大いに利用して下さい。

4・1

F/ファイルメニュー　GRPH+F

ソースリストなどのファイルを管理・操作する

N/新規

現在の編集中のソースリストが変更されて保存されていないならば、保存するかしないかの確認を求めて破棄し、新しくビューウィンドウ（プログラムソースを読み込むところ）を作る。ファイル名を指定していない場合には、ファイル名を要求してくる。テキストファイルならば、どんなファイルでも編集可能。このコマンドを実行すると、サブファイルも同時に破棄されるので注意。

O/読込

指定されたファイルを読み込む。ファイルの指定は、ダイアログボックスを用いて指定。

N/ファイル名	読み込みたいファイルを書き込む。拡張子の＊.BASは省略可能。ワイルドカードの使用可。ダイアログボックスに表示されるファイルリストにカーソルを移動するには、TABキーを一回押し、カーソルキーで目的のファイルにカーソルを合わせ⏎キーを押す。

M/追加

エディット中のファイルの、カーソルのある行に指定のファイルを追加する。

ダイアログボックスの［N/ファイル名：］に直接ファイル名を記入するか、TABキーを1回押し、ファイルリストにカーソルを移動してファイルを選ぶ。

S/保存

ソースリストを、ディスク上に保存する。保存するファイル名は変更できないので、ファイル名を変更する場合には[A/名前を変えて保存...]を使う。また、[S/保存] には、保存形式というオプションがあります。

Q/標準ファイル	.BASという拡張子のついたQuick BASIC専用のファイルとして保存する。読み込む速度は早くなるが、他のエディタやワープロなどでは、読み込み不可能となる。
T/テキストファイル	ASCIIファイル、またはテキストファイルとして保存する。ドキュメント、インクルードファイルは必ずこの形式で保存される。

A/名前を変えて保存

ソースリストのファイル名を変更して、ディスクに保存する。[S/保存]同様に保存ファイル形式を選択できる。

V/すべて保存

現在オープンされているファイルを、全てディスクに保存する。画面に表示されていないような、サブファイルももちろんその対象になる。

C/サブファイルの作成

新規にモジュール・インクルードファイル・ドキュメントを作成するときに指定する。ダイアログボックスに、作成するファイルのファイル名とファイルのタイプを指定する。

M/モジュール	モジュールとは、プログラムを構成する要素の最小単位で、Quick BASICでプログラムとして扱えるファイルのこと。 ファイル名には自動的に“*.BAS”という拡張子がつく。
I/インクルード	インクルードファイルとは、$INCLUDEメタコマンドで指定されるテキストファイルのこと。標準のQuick BASICの形式では扱えない。テキスト形式でファイルがオープンされる。拡張子などは、ユーザーが管理する必要がある。
D/ドキュメント	普通の文書などを作成する場合に指定する。ドキュメントとしてオープンされたファイルを編集する場合には、構文チェックなどはしない。

L/サブファイル常駐

メモリにモジュール・インクルード・ドキュメントを読み込む。読み込まれたファイルはメモリに常駐する。1つのモジュールを読み込んだとき、他のモジュールが付属しているような場合には、そのモジュールも同時に読み込む。

読み込んだモジュール間の移動は［f•2 SUBファイルの表示］などを使用する。

N/ファイル名：	読み込みたいファイル名を指定。マウスやTABキーで、ファイルリストの中からの指定もできる。
形式：	読み込むファイルの形式を指定します。形式については、C/サブファイルの作成を参照してください。

U/サブファイル解放

メモリに読み込まれているモジュール・インクルード・ドキュメントを、メモリから解放する。

P/印刷

現在読み込んでいるモジュール・インクルード・ドキュメントの印刷を行なう。次のオプションに注意。

S/指定範囲	アクティブウィンドウの指定範囲を印刷。範囲の指定は[↑][↓]で印刷したいリストの先頭行にカーソルを移動し、[SHIFT]+[↑][↓]で印刷したいリストの最終行を指定。この状態から（画面は反転する）[GRPH]+F→P→Sを実行する。印刷終了後、反転していた画面を元に戻すには、カーソルキーを押す。
W/アクティブウィンドウ	アクティブウィンドウの内容を印字。モジュールでないファイルが表示されていても印字を行なう。
M/カレントモジュール	アクティブウィンドウのモジュール全体を印字。アクティブウィンドウに表示されていないSUB、FUNCTIONプロシージャがあっても、アクティブウィンドウのモジュールの一部ならば印字する。
A/全体	表示されているいないに関わらず、メモリに読み込まれている全てのファイルを印字。

D/MS−DOSコマンド

一時的にMS-DOSに制御を移す。＊.EXEモデルのプログラムを作成して動作確認をとる場合などに必要。MS-DOSからエディタに復帰する場合には、コマンドラインからEXITと入力する。

X/終了

Quick　BASIC（エディタ）を終了し、完全にMS-DOSに制御を移す。再度Quick BASICに戻る場合は、あらためてQuick BASICを起動する。

4・2

E/編集メニュー　GRPH＋E

ソースリストを編集する

U/元に戻す（GRPH＋BS）

変更を加えた行を変更前に戻す。行だけに機能するので、全てのソースリストを元に戻すわけではない。

T/カット（SHIFT＋DEL）

範囲指定を行なった部分を画面から削除し、クリップボード（編集用の一時記憶領域）に取り込む。SHIFT＋↑ ↓で範囲指定する。クリップボードに取り込んだ内容を画面に張り付ける場合には、張り付ける位置にカーソルを移動し［P/ペースト］を実行。

C/コピー（CTRL＋INS）

指定範囲を行なった部分をクリップボードに取り込む。クリップボードに取り込んだ内容を画面に張り付ける場合には、張り付ける位置にカーソルを移動し［P/ペースト］を実行。

P/ペースト（SHIFT＋INS）

［T/カット］［C/コピー］で、クリップボードに取り込んだ内容を画面に張り付ける。張り付ける位置は、現在のカーソル行。

E/削除（DEL）

指定部分を削除する。クリップボードに削除した内容を記憶しないため、削除された文字を復活することは出来ない。注意。

S/新規SUB...

新しいSUBプロシージャを定義する。ウィンドウは別に開かれるが、実際はメインルーチンの一部となる。ただし、アクティブウィンドウ上から

SUB　SUBプロシージャ名↵

と入力しても、自動的にSUBプロシージャ用のウィンドウが開く。

F/新規FUNCTION...

新しいSUBプロシージャを定義する。ウィンドウは別に開かれるが、実際はメインルーチンの一部となる。ただし、アクティブウィンドウ上から

FUNCTION　FUNCTIONプロシージャ名↵

と入力しても自動的にFUNCTIONプロシージャ用のウィンドウが開く。

Y/構文チェック

自動構文チェック機能のON/OFFを設定。自動構文チェック機能とは、エラーを引き起こす可能性のある文を入力時点で検出すること。

4•3

V/表示メニュー　GRPH＋V

ディスプレイ表示の操作をする

S/SUB一覧...（f•2）

メモリに読み込まれているSUB、FUNCTIONプロシージャの一覧表を表示。また、一覧表が表示されている時にプログラム間の移動が行なえる。

C/プログラム選択：	表示•編集のできるプログラムを変更できる。希望のプログラムにカーソルを合わせてリターンキーを押す。ただし[W/アクティブウィンドウ]に設定されていること。
W/アクティブウィンドウ	アクティブウィンドウのプログラムを指定。メニューの外枠が太字になっている時に［C/プログラム選択：］でファイルを選択すると、そのファイルがアクティブウィンドウに指定される。
S/分割ウィンドウ	画面を上下に分割。アクティブウィンドウが特に設定されていない場合、現在アクティブウィンドウにあるファイルが、2分割される。それぞれのウィンドウに別のファイルを表示させたい場合には［W/アクティブウィンドウ］を実行する。
M/移動	SUB、FUNCTIONプロシージャを、別のモジュールで使えるように移動する。このコマンドを実行する前に［C/プログラム選択：］で設定しておくこと。
D/削除	SUB、FUNCTIONプロシージャを削除。SUB、FUNCTIONプロシージャは、Quick BASICで独自に管理されているため、本コマンドを使って正規に削除する。あらかじめ、削除するプロシージャに[C/プログラム選択：］でカーソルを合わせておく。

E/次のSUB（SHIFT+f・2）

アクティブウィンドウに表示されているファイルを切り替える。切り替えられるファイルは［S/SUB一覧...］で表示できるファイル。

P/画面分割

画面を上下に2分割する。すでに画面が2分割されている場合には、1画面表示にする。

N/次のステートメント

次に実行されるステートメントを表示する。

U/実行画面（f・4）

MS-DOSを疑似的に作り出してプログラムの実行を確かめるための画面。プログラムを実行させ、任意の位置で停止させているときの画面も表示可能、またプログラムが完全に終ってからでも表示可能。

I/インクルードファイル編集

現在読みこんであるインクルードファイルを表示・編集する。

L/インクルードファイルの表示

インクルードファイルを表示する。

O/オプション...

画面のバックグラウンドのカラー・文字色・ブレークポイントの設定された行の色など、Quick BASICの表示形式を変更・保存する。目的の制御行（例えば、カレントステートメントの色を変える場合には、カレントステートメントが目的の制御行）にカーソルを移動し、スペースバーで色を変更し、⏎キーで決定。設定された内容は保存され、次のQuick BASICを起動した場合にも有効。

N/テキスト	テキスト画面の文字色・背景色を決定する。
C/カレントステートメント	カレントステートメント（バー状の印）の文字色・背景色を決定する。
B/ブレークポイント行	ブレークポイント行（バー状の印）の文字色・背景色を決定する
S/スクロールバー	画面の右と下の行にある、スクロールバーを、表示する／しないを設定する。
T/タブ間隔：	TABキーを押したとき、カーソルを移動させるときの量を決定する。デフォルトは4。

4・4

S/検索メニュー　GRPH＋E

文字列の検索と置換、およびエラー行を検索

F/検索...

現在読み込んでいるテキストやSUB・FUNCTIONプロシージャの中から、指定の文字列を探す。次のオプションがある。

F/検索文字列	検索する文字列を書き込む。検索したい文字位置にカーソルが移動している場合には、検索文字列が表示される。
1/アクティブウィンドウ	アクティブウィンドウ内で検索を行なう。
2/カレントウィンドウ	カレントウィンドウ内で検索を行なう。
3/全体	読み込まれているモジュール、インクルード、ドキュメントファイル全てを対象に検索を行なう。
M/大小文字区別	大文字・小文字の区別をチェックする。
W/全体一致	完全に一致する文字列を検索する。検索文字列の前後の文字も厳重にチェックするので、文字列中にある検索文字などは検索しない。

S/指定文字列検索（CTRL＋¥）

ソースリスト中から指定した文字列を検索する。検索したい文字列にカーソルを移動してから、このコマンドを実行する。

なお、オプションは［F/検索...］と同じ。

R/次を検索 (f･3)

直前に実施した[S/指定文字列検索][C/置換コマンド]で検索の対象となった文字列の次に現れる文字列を検索。

C/置換... (CTRL＋Q＋A)

指定の文字列を探し、指定の文字列に置き換える。次のオプションがある。

F/置換前	置き換える前の文字列を指定。
T/置換後	置き換える文字列を指定。
V/逐次確認	検索文字列を置き換えるかどうかを確認しながら置換する。
C/一括置換	確認をせずに、自動的に置換を開始する。

※その他のオプションについては[F/検索]を参照。

L/ラベル...

BASICでラベルと認められた文字列を検索する。

4・5

R/実行メニュー　GRPH＋R

コンパイルおよびインタプリタによる実行

S/スタート（SHIFT＋f・5）

現在読み込んでいるプログラムを、インタプリタで実行する。

R/リスタート

トレースできるように変数・カレントステートメントを初期化する。

N/続行（f・5）

初期化が済んでいれば、先頭から実行。シングルステップトレースやブレークポイントで停止している場合には、次の行から実行。プログラムが変更され、結果に影響のある場合にはメッセージを出してから実行を継続。

C/COMMAND$入力...

COMMAND$関数の返す文字列を定義する。プログラム呼び出しに使用する入力パラメータを与えるコマンド。

X/EXEファイルの作成

ソースリストをコンパイル、リンクして単独で実行できるプログラム（＊.EXE）にする。以下のオプションがある。

N/EXEファイル名	＊.EXEファイルにするファイル名を入力する。
X/ランタイム分離型EXEファイル	このランタイム分離型EXEファイルを指定して作成されるEXEファイルは、これ自身では動作できず、Quick BASICのランタイムルーチンを別に起動する必要がある。ただし、プログラムが非常にコンパクトにまとまる。また、ランタイムルーチンはいったん起動すると、メモリに常駐し他のQuick BASICのプログラムでも使用可能
A/独立型EXEファイル	単独で実行できるEXEファイルを作成する。ランタイム分離型と比べると、ややプログラムサイズが大きくなる。
D/デバッグコード	デバッグコードを、.EXEの中に付加する。このコードを含んだプログラムは、＊.EXEとなってもデバッグが可能となる。

L/ライブラリ作成...

自作の関数などを登録するクイックライブラリを作成する。

N/Quickライブラリファイル名：	ライブラリとするファイル名を指定する。
D/デバッグコード	ライブラリ作成後の状態を指定する。

M/メインモジュールの設定...

メインとなるモジュールを変更、設定する。

4・6

D/デバッグメニュー　GRPH+D

プログラムの追跡・ブレークポイント設定などのデバッグ操作

A/ウォッチの追加...

ウォッチウィンドウに表示する変数、式を設定する。

W/ウォッチポイント...

条件式を設定し、条件式が真となるまで実行を続ける。

D/ウォッチの削除

ウォッチ式（変数、式）をウォッチウィンドウから削除する。

L/全ウォッチ削除

全てのウォッチ変数（変数、式）をウォッチウィンドウから削除する。

T/トレース

プログラムを実行する時、一定のゆっくりしたスピードでプログラムを実行（PRINT文などの画面制御のステートメントを実行すると、実行画面とエディタが交互になるため、若干見にくくなる)。トレースの実行を止めるには、CTRL+Cもしくは、STOPキー。［T/トレース］にチェックマークがあるときに選択すると設定が解除され、マークがないときに選択すると設定される(トグルスイッチ)。

H/ヒストリ

実行したステートメントの実行状態を記憶する(20行まで)。記憶した内容は SHIFT＋f･8（前に移動）、または SHIFT＋f･10（後ろに移動）で確認。トグルスイッチになっているので［H/ヒストリ］にチェックマークがあるときに選択すると設定が解除され、何もマークがないときに選択すると設定される。

B/ブレークポイント（f･9）

現在カーソルのある行をブレークポイントに設定／解除する。カーソル行が、すでにブレークポイントに設定されていれば設定を解除し、設定されていなければブレークポイントとして設定する。

ブレークポイントが設定されると、プログラムの実行を一時中断し、これをウォッチウィンドウと合わせて使えば、変数の内容の変化や流れなどが詳しく分かり、デバッグに役立つ。ブレークポイントはいくつでも設定できる。ブレークポイントの色を変更するには［V/表示メニュー］→［O/オプション］→［B/ブレークポイント行］で設定する。

C/全ブレークポイント解除

設定されているブレークポイントを全て解除する。

S/カレントステートメントの設定

カーソル行の次からプログラムを実行するように設定し、任意の位置からプログラムを実行できる。

4・7

H/ヘルプメニュー　GRPH＋H

ヘルプの表示

G/全般情報...（f・1）

全般的なヘルプ情報を表示。［N/次のページ］で、次のヘルプページを表示。［P/前のページ］で前のヘルプページを表示。［K/キーワード］で特定のキーワードに対するヘルプ情報を表示。

T/キーワード：（SHIFT＋f・1）

Quick BASICのキーワードや関数についての情報を表示する。ソースリスト中のキーワードを参照するには、カーソルを参照させたい文字列に移動後、このコマンドを実行する。［T/キーワード］に登録されている場合には該当する語句を探すことなく、ヘルプ画面を参照できる。なお、［G/全般情報］→［K/キーワード］でも参照できる。

C/ヘルプを閉じる

現在開いているヘルプ画面を閉じる（ESCでも同じ）。

4・8

C/関数メニュー　GRPH＋C

ネストした関数のトレース（追跡）・表示

ネスト（関数が関数を呼び出す）した関数が、それまで呼び出した関数のリストを表示する。表示されたリストを使い、ある特定の関数まで処理を戻せる（↑ ↓で関数を選択）。

第5章

Quick BASICの基礎

いよいよ、Quick BASICでプログラミングを開始していきましょう。

Quick BASICは、実に様々なことが行なえます。画面制御やファイル管理はもちろんのこと、グラフィックステートメントを用いれば、自分専用の図形プロセッサさえも作成可能です。

しかし、まずはQuick BASICの基礎をしっかりと学んでおきましょう。

この章では、PRINT文とINPUT文を使って、Quick BASICでのプログラミングの基礎の基礎について解説します。

では、Quick BASICを起動して、ソースを入力、実行していきましょう。ソースを入力したら、必ず保存しましょう。.BASのついたソースリストがあれば、Quick BASICでは、いつでもインタプリタ（f･5キー）で実行できますから、学習段階では、すべてのソースをEXEファイルにする必要はありません。

5・1
PRINT文を使って画面出力

画面に文字を表示しよう

1　PRINT文の使い方

PRINT文で、１行を画面表示する

第３章で

PRINT "QUICK BASIC"

というプログラムを書き、実行してみました。これがPRINT文なのです。PRINT文とは、画面に指定した文字（メッセージ）を表示するため命令です。

このような命令のことを、Quick BASICでは「ステートメント」と呼びます。

●●PRINT文の書式●●

PRINT "メッセージ"　　←メッセージ部分が画面に表示される

●●PRINT文の使い方●●

◉ソースリストPRG-NO2

```
PRINT "Today is fine."
```

「""」で表示したい文字を囲む

表示したい文字。大文字、小文字などがまじってもよい

このプログラムは、“Today is fine.” と表示されるはずです。

f･5キー（インタプリタによる実行）を押してみましょう。

●インタプリタによる実行結果

```
Today is fine.

何かキーを押してください
```

大丈夫でしたね。

このように、PRINT文では「” ”」で囲まれた中のメッセージを表示します。

これがもっとも基本的なPRINT文の使い方です。

PRINT文による複数行の画面表示

PRINT文で、1行のメッセージを出力する方法は分かりました。では、2行、3行と複数行にわたってメッセージを出力してみましょう。

複数行にわたってメッセージを出力する場合、PRINT文を出力する数だけPRINT文を書きます。

●●ソースリストPRG-NO3.BAS●●

```
PRINT "QUICK BASIC"
PRINT "Today is fine."
```

◉インタプリタによる実行結果

```
QUICK BASIC
Today is fine.

何かキーを押してください
```

2 色を付けてメッセージ出力（COLOR文）

ここまでのメッセージ出力は、文字は白、背景色が黒と決っています。

これだけでは、ちょっと味気ないと思えば、COLOR文を使って文字や背景に、コンピュータの許す範囲内で色を付けられます。

●●COLOR文の書式●●

COLOR 文字色，背景色

文字色や背景色は全て、次のような番号で管理されています。

◉文字色・背景色の色番号

色	通常の色指定	背景色と反転	通常の色の点滅指定	反転時の点滅指定
黒	0	8	16	24
青	1	9	17	25
緑	2	10	18	26
水色	3	11	19	27
赤	4	12	20	28
紫	5	13	21	29
黄色	6	14	22	30
白	7	15	23	31

これらの数字をCOLOR文の後に書き込むだけで、文字色を好みの色にしたり、点滅をさせたりすることができます。

ただし、背景色の点滅はできません。背景色で設定できるのは、通常の色指定だけですので、0～7までの数値だけです。もし背景色にこれら以外の数値を指定すると、

引数が許される範囲ではありません（ERR=5）

というエラーメッセージが出力されます。

さて、簡単なCOLOR文を、実際に使ってみることにしましょう。

◉ソースリストPRG-NO4.BAS

```
COLOR 5
PRINT "QUICK BASIC"
COLOR 6
PRINT "Today is fine."
COLOR 11
PRINT "PRE STAGE"
```

◉インタプリタによる実行結果

```
QUICK BASIC ──────紫
Today is fine.────黄色
PRE STAGE ────────水色の背景が反転、文字が黒となる

何かキーを押してください
```

この本では、カラーの文字表示をお見せできませんが、みなさんのディスプレイには、カラフルなメッセージが出力されたはずです。「文字に色を付ける」などの作業は、プログラムとして実質的なことは何も行なっていませんが「エラーメッセージを赤で表示して、キー入力を促す」などのような場合、文字に色を付けたくなることがあります。

このようなときに、COLOR文を使って下さい。

また、背景色を瞬時に交換することで、画面をフラッシュさせるといったこともできます。

3 画面消去（CLS文）

画面に、以前実行したプログラムの実行結果が残っているような場合、目障りだったり、うっとうしかったりします。

ここで、画面をきれいに掃除する方法を学んでおきましょう。

画面をクリアするには、CLS文を使います。

●●CLS文の書式●●

CLS オプション

画面をクリアしたいところで、CLS文を書き込むことにより、画面のクリアができます。

また、オプションの数値を操作すれば、グラフィック画面（線などを引く画面）だけクリアするとか、テキスト画面（通常文字が出力される画面）だけをクリアするなど、様々なことに応用できます。

ただ単にCLS文だけで、オプションを付けなければ、グラフィック画面、テキスト画面の双方共クリアします。

グラフィックだのテキストだのと、最初は何をいっているか分からないと思いますが、ここでは取り合えず「CLS文は画面クリアのためのステートメント」ということだけを覚えてください。

◉ソースリストPRG-NO5.BAS

```
CLS
COLOR 5
PRINT "QUICK BASIC"
COLOR 6
PRINT "Today is fine."
COLOR 11
PRINT "PRE STAGE"
```

◉インタプリタによる実行結果

```
                    ←―――以前あったメッセージが全てなくなっている点に注目

QUICK BASIC  ←―――紫
Today is fine.  ←―――黄色
PRE STAGE  ←―――――水色の背景が反転、文字が黒となる

何かキーを押してください
```

CLS文は、実に頻繁に使われます。

メニューを作成したりする場合、次のメニューを表示したいときに、いったん画面をクリアする必要がありますが、このようなときにCLS文を使います。

4　メッセージ位置の指定（LOCATE文）

今までの方法だと、メッセージの位置を自由に指定することはできません。

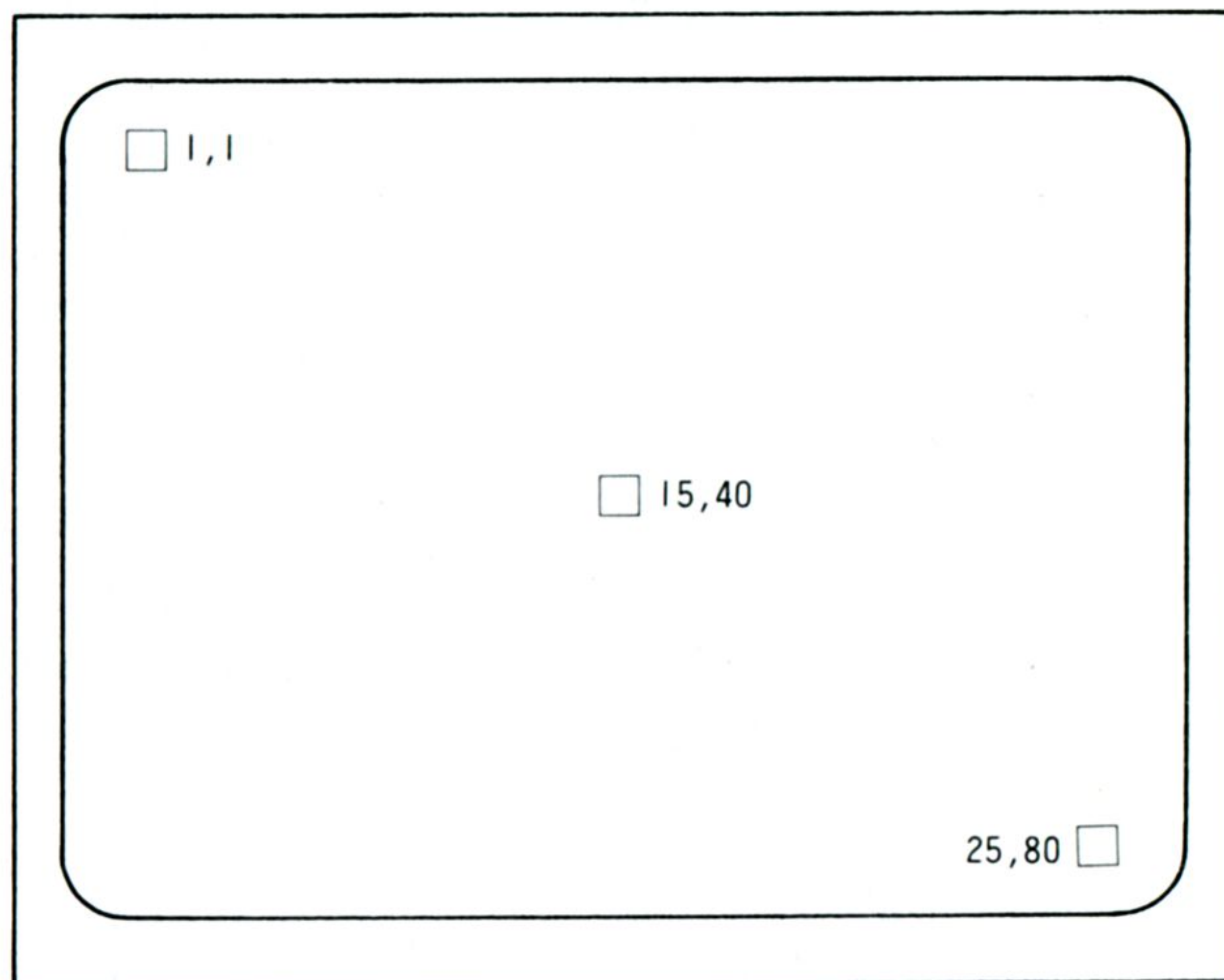
例えば、画面のちょうど中央にメッセージを出力したいとか、画面の右下にメッセージを出力したいとかいうことができません。

こんな時には、LOCATE文でメッセージ位置を自由に指定できます。

●●LOCATE文の書式●●

LOCATE 行，桁

行とは、画面の上から数えて何行目かということです。
桁とは、左から数えて何桁目にするかということです。
また、画面の行・桁は、次のように決っています。

◉画面の行と桁の位置関係

ここで示されているように、画面の左上の角がLOCATE 1,1となり、画面の右下の角がLOCATE 25,80です。

この数値以上は、行・桁ともに与えることはできません。つまり、通常の文字出力はLOCATE 1,1～LOCATE 25,80までと決っているのです。

実際にプログラムを打ち込んでみて、画面の指定位置にメッセージを出力してみましょう。

◉ソースリストPRG-NO5.BAS

```
CLS
LOCATE 15,40
PRINT "QUICK BASIC"
```

◉インタプリタによる実行結果

```
                                QUICK BASIC



何かキーを押してください
```

LOCATE文により、画面のちょうど中央を指定しているので、メッセージが画面中央に出力されています。LOCATE文が確かに位置を、指定していることが分かります。

5•2

INPUT文を使ってキーボード入力

キーボードから入力したキーを画面に表示

キーボードから入力したキーを判断させるには、INPUT文を使うのがもっとも簡単です。

ここでは、INPUT文を使ってキーボード入力を理解しましょう。

●●INPUT文の書式●●

INPUT [; もしくはなし] "メッセージ" [, もしくは ;] 変数

"変数"という言葉が出てきました。これは一体なんでしょうか。

変数とは、任意の数値などを格納しておくための箱なのです。

変数については、次の章で詳しく解説しますが、ここでは取り合えず、「任意の数などを入れる箱」と解釈しておいて下さい。

また、INPUT文は、オプションの使い方で様々な使用方法があります。ここでは、もっとも簡単な使い方を見てから、実戦的な使い方までを知ることにします。

◉ソースリストPRG-NO7.BAS

```
INPUT X ←―――キーボートにより変数Xに格納
```

◉インタプリタによる実行結果

```
? 100
```

INPUT文だけを実行すると、画面には「？」が出力され、キーボードからの入力を促されます。

ここでは“100”と入力してみしましたが、では入力した数値はどこにいってしまったのでしょうか。

次に、入力を行なった文字を表示させて、確かに入力されたかの確認をとってみましょう。

◉ソースリストPRG-NO8.BAS

```
INPUT X
PRINT X  ←――変数の内容の表示
```

◉インタプリタによる実行結果

```
? 10
 10  ←――変数の内容を表示させると確かに“10”が入っていた
```

INPUT文によって変数Xに10を入れ、その結果がPRINT文によって表示されています。

ここで注意していただきたいのですが、ここ扱った変数は数値型といって、数値しか受け付けない変数です。したがって、文字（ABCなど）を“？”の次に入力すると

再入力してください

とメッセージされます。

メッセージ出力を伴ったキーボード入力

次に、メッセージ出力を伴った、キーボード入力を行なってみることにしましょう。

キーボード入力を行なう場合、普通はなんらかのメッセージを出力して、入力を促します。

ここではその方法について解説します。

◉ソースリストPRG-NO9.BAS

```
INPUT "Please data ";X
PRINT X
```

注意（「;」を指す）

◉インタプリタによる実行結果

```
Please data ? 1500
 1500
```

INPUT文は、先ほどの理由などからメッセージ出力を伴うことが多いので、最初からメッセージ出力を行なえるように工夫されています。

メッセージを出力させる場合には、PRINT文などと同じようにメッセージを「”　”」で囲んでやります。

これでメッセージ付きでキーボード入力を受け付けられますが、注意したいのが、変数とメッセージの区切りです。

ソースリストPRG-NO9.BASのように「；」（セミコロン）で区切ります。

INPUT文の「?」出力を消す

これで、一通りINPUT文について把握できたと思いますが、INPUT文を使うと必ず出力される「？」を消してみましょう。

場合によっては、目障りに感じられることがあります。

「？」を消すには、メッセージと変数の区切りを「；」(セミコロン)から「,」(カンマ) に変更します。

試しに、先ほどのソースリストを使って動作を確認してみましょう。

◉ソースリストPRG-NO10.BAS

```
INPUT "Please data ",X
PRINT X
```

"；"を","に変更

◉インタプリタによる実行結果

```
Please data 2
 2
```

第6章

変数の基礎

この章では、変数について学びます。

すでに、変数を使ったプログラミングをしてみましたが、まだ、本質的な意味を理解していないと思います。

ここでは、変数の考え方から解説します。

変数には、文字型変数と数値型変数がありますが、実際に変数を使ったプログラムを作って、その内容や使い方を理解してください。

どのようなプログラミング言語でも変数の概念は、大切なものです。

みなさんも変数を自由に扱えるようになって下さい。

6・1

変数の概念

全てのプログラミングの根本的な考え方

変数とは、すなわち

変動する値を入れておく入れ物

なのです。

プログラミングにおいてこの考え方は重要で、全てのプログラミング言語はこの考え方に基づいてプログラミングを行ないます。

この変数という入れ物には、数値だけでなく文字も入れられます。

しかし、数値を入れる入れ物と文字を入れる入れ物は明確に区別しなくてはなりません。

また、変数は代入を行なうためにも使われます。

変数は、本書でもたくさん使っていきますので、徐々に覚えていくことにしましょう。

まず、次のことを覚えておいて下さい。

●変数名は、40字以内で自由に名前を付けられる
●変数名の最初はかならず文字であること
●ステートメントや関数で使っている文字列は使えない
●数値を扱う変数は数値型変数
●文字を扱う変数は文字型変数

6•2

数値型変数

変数名に何もつけずに代入

1　数値型変数の使い方

数値型の変数は、変数名に何も付けずに代入を行ないます。

簡単なリストを通して、数値型の変数を理解しましょう。

◉ソースリストPRG-NO12.BAS

```
A = 2
B = 3

PRINT A
PRINT B
      └──────── 変数を表示させる
```

◉インタプリタによる実行結果

```
2
3
```

確かにそれぞれの変数の内容が表示されました。

ここでは、変数の内容とメッセージを同時に表示させていますが、PRINT文にはこんな機能もあります。

今までPRINT文を使ってメッセージを表示させていたとき「""」で囲んでいましたが、変数を表示させるときには「""」がいらない点に注意してください。

また、変数に数値を代入する場合の方向に注意してください。
次のように、BASICの代入では、右から左に行なわれます。

◉変数への数値の代入方向

```
A = 3000     ←右（3000）の数値が、左の変数Aに代入される
```

2　数値型変数の扱う桁数と範囲

さて、ここで扱った数値型の変数は、扱える桁数には限界があります。扱える桁数によって宣言が必要になります。

■宣言をしない場合

扱える範囲　　-9,999,999～＋9,999,999
［例］X = 100

■変数の末尾に％を付ける

扱える範囲　　-32,768～＋32,767
［例］X% = 12000

■変数の末尾に＆を付ける

扱える範囲　　-2,147,483,648～＋2,147,483,647
［例］X& = 1000000000

ここでは、小数点を含まない整数での桁数と範囲を示しました。
小数点を含む正確な代入などは「固定小数点」「浮動小数点」という方法を用います。
「固定小数点」「浮動小数点」などは、もう少しプログラミング技術が向上してから学んで下さい。本書では、整数値だけを扱います。
さて、これまで、いくつか使ってきた数値型の変数は、実は、最大7桁までしか扱えなかったのです。
試しに、7桁を越える数値を代入してみることにしましょう。

◉ソースリストPRG-NO13.BAS

```
CLS
A = 10000000  ←─── 8桁の数値を代入
PRINT "A ="; A
```

◉インタプリタによる実行結果

```
A = 1E+07
```

表示させるつもりの“10,000,000”は表示されず、妙な値が表示されました。

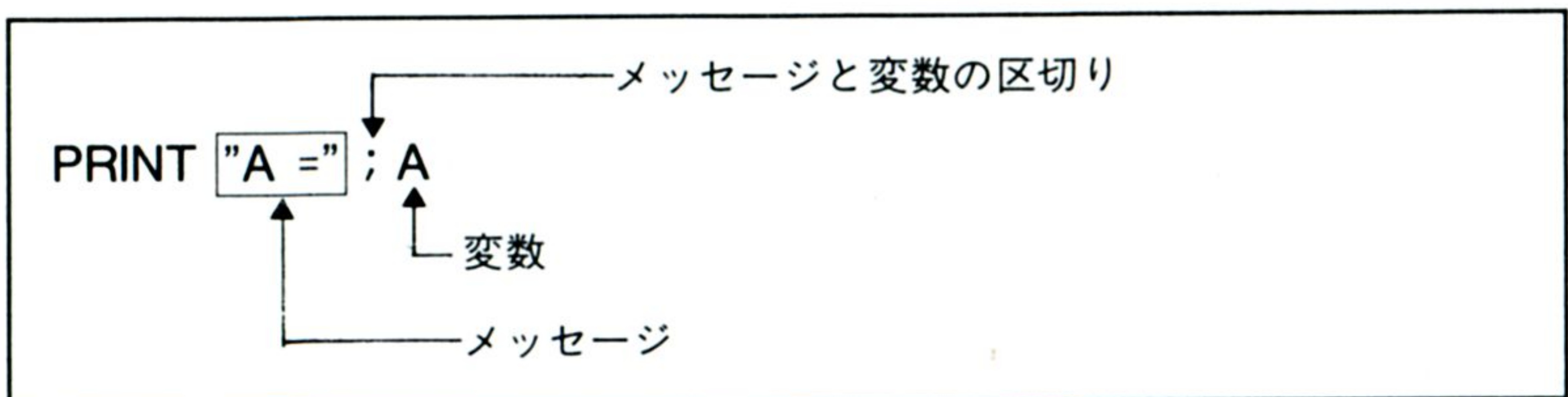
これは指数と呼ばれ、実際に変数Xに数値を入れているものの、それを扱いきれなくなったために起こる現象です。

実際にはちゃんと代入されていますが、Quick-BASICで限界のところまできてしまったので、このような表示となってしまいました。

また、ここでは、もう一つのことを行なっています。PRINT文では、メッセージと変数を同時に表示させることができるので、その機能をここで使っています。

●●PRINT文でメッセージと変数を表示させる●●

```
PRINT "A =" ; A
```

- "A =" ：メッセージ
- ; ：メッセージと変数の区切り
- A ：変数

メッセージと変数を同時に表示させるには、それぞれの約束ごと(“”で囲むなどの）を満たした上で、「；」で区切ればちゃんと表示されます。

また、同時に表示させるメッセージと変数の順番はありません。どちらが最初にきても、また連続しても構いません。

なお、区切りのセミコロンをカンマに変更すると変数とメッセージの間に特定の桁数、空白を開けます。

もう少し、例を挙げてみましょう。

◉ソースリストPRG-NO14.BAS

```
CLS
A% = 10000B
& = 1000000
PRINT "A ="; NO1%          ← 記号に注意
PRINT "B ="; NO2&
```

◉インタプリタによる実行結果

```
A = 10000
B = 1000000
```

変数を代入し、変数を表示するなどの場合に、もし記号を忘れると、変数Aと変数A%とは別の変数として扱われるので注意が必要です。

◉変数Aが変数A%とは別の変数として扱われる例

```
CLS
A% = 10000
B& = 1000000
PRINT "A ="; A          ← 記号を忘れている
PRINT "B ="; B
```

◉インタプリタによる実行結果

```
NO1 = 0 ┐
NO2 = 0 ┘── 記号を忘れたので別の変数として扱われた
```

6•3

文字型変数

変数名の最後に$、代入する文字は" "でくくる

1　文字型変数の使い方

文字型の変数は、

変数名の最後に「$」マーク

を付けて代入を行ないます。また、

代入する文字は「""」(ダブルクォート) でくくる

必要があります。

この2点だけが数値型変数との違いです。はじめは、忘れてしまうことが多いので注意しましょう。こんな間違いでも、長いプログラムでなかなか見つからないと苦労します。

◉ソースリストPRG-NO16.BAS

```
CLS
A$ = "KUDOU"
B$ = "SHIZUKA"
PRINT "A$ = "; A$
PRINT "B$ = "; B$
```

「$」マークに注意

◉インタプリタによる実行結果

```
A$ = KUDOU
B$ = SHIZUKA
```

2　文字型変数の扱える最大長と種類

数値型の変数にも桁数などの制限があったように、文字型の変数でも制限があります。要約すると次のようになります。

■可変長文字列

意　味　長さが定められていない文字列
最大長　32,767文字以内

■固定長文字列

意　味　文字数を宣言してから使う文字列のこと
最大長　32,767文字以内

「可変長文字列」「固定長文字列」共に扱える文字数は同じですが、処理が若干違います。可変長の場合には、扱う文字の長さが後から変わっても問題ありませんが、固定長の方は最初に決めた文字数しか代入できません。通常は可変長を使います。

これらの概念は、ファイル操作をするときに使いますが、ここでは概念だけを理解しておいて下さい。

文字型変数を確認する意味で、簡単なリストを作成し、実行してみましょう。

◉ソースリストPRG-NO17.BAS

```
CLS
INPUT "Please your name : ", NAME$
PRINT "Your name is "; NAME$
```

◉インタプリタによる実行結果

```
Please your name : KUDOU SHIZUKA
Your name is KUDOU SHIZUKA
```

第7章
Quick BASICの演算

　コンピュータの演算には、

四則演算
比較演算
論理演算

があります。
　また、コンピュータに特長的な演算として

文字列の演算

があります。
　これらの演算は、コンピュータにおける演算の初歩中の初歩です。一見難しいように見えますが、実際に簡単なプログラムを作ってみれば、その概念は容易に把握できるはずです。
　簡単なプログラミングを通して、演算について学んでいきましょう。

7・1

四則演算

引き算、足し算、掛け算、割算

1　四則演算の概念

四則演算とは、御存知のように

－（引き算）
＋（足し算）
×（掛け算）
÷（割り算）

のことです。このー、＋などの記号を演算子といいます。

四則演算の演算子

足し算（演算子：＋）
AとBを足してCに答えを代入する（[例] C=A＋B)

引き算（演算子：ー）
AからBを引いてCに答えを代入する（[例] C=A-B)

掛け算（演算子：＊）
AとBを掛けてCに答えを代入する（[例] C=A＊B)

割り算（演算子：／）
AをBで割ってCに答えを代入する（[例] C=A/B)

剰余演算（演算子：MOD）
AをBで割ってCに余りを代入する（[例] A MOD B)

べき乗（演算子：^）
AのB乗した答えをCに代入する（[例] C=A^B)

このように×、÷には特別の記号を使います。掛け算には「＊」を、割り算には「／」を代用して実際の計算に用います。これは、×、÷の記号がキーボードにないからです。

また、通常の四則演算の他に、BASIC固有の計算があります。

剰余演算とべき乗です。

剰余演算とは「割った余り」を算出する計算方法で、実際のプログラミングには欠かせない演算方法です。また、べき乗は「○○の３乗」などの表現をコンピュータに行なわせるときに使います。

2　数値の四則演算

先ずは、理解しやすい四則演算を行なってみましょう。

◉ソースリストPRG-NO18.BAS

```
CLS
A = 3
B = 2
PRINT "A ="; A; " B ="; B
PRINT

C = A + B
PRINT "A + B ="; C
C = A - B
PRINT "A - B ="; C
C = A * B
PRINT "A * B ="; C
C = A / B
PRINT "A / B ="; C
C = A ^ B
PRINT "A ^ B ="; C
C = A MOD B
PRINT "A MOD B ="; C
```

変数Cにそれぞれの計算結果を代入

●インタプリタによる実行結果

```
A = 3 B = 2

A + B = 5
A - B = 1
A * B = 6
A / B = 1.5
A ^ B = 9
A MOD B = 1
```

上のリストは、それぞれの計算を変数Cに代入して、メッセージと共に表示させましたが、これと同じことをスッキリとまとめて、次のようにも書くことができます。

●ソースリストPRG-NO19.BAS

```
CLS
A = 3
B = 2
PRINT "A ="; A; " B ="; B
PRINT

PRINT "A + B ="; A + B
PRINT "A - B ="; A - B
PRINT "A * B ="; A * B
PRINT "A / B ="; A / B
PRINT "A ^ B ="; A ^ B
PRINT "A MOD B ="; A MOD B
```

計算結果をそのまま表示することも可能

◉インタプリタによる実行結果

```
A = 3 B = 2

A + B = 5
A - B = 1
A * B = 6
A / B = 1.5
A ^ B = 9
A MOD B = 1
```

3　文字列の演算

今度は、文字列の演算を行なってみましょう。

文字列の演算は、足し算はむろんのこと、引き算などもできます。しかし、引き算などはやや高度な技術が伴うため、足し算だけが文字列の演算として使われることが多いようです。

◉ソースリストPRG-NO20.BAS

```
CLS
INPUT "First name  : ", NAME1$
INPUT "Second name : ", NAME2$
PRINT "First name = "; NAME1$; "  Second name = "; NAME2$
PRINT "Your name is "; NAME1$ + NAME2$
```

◉インタプリタによる実行結果

```
First name  : Yui
Second name : Asaka
First name = Yui  Second name = Asaka
Your name is YuiAsaka
             └──── 確かに連結された
```

First nameで入力した部分が“Yui”でSecond nameで入力した部分が“Asaka”です。

ご覧の通り、たしかに連結されました。

7・2

比較演算子

式の左右の値を比較して判断する

1 比較演算の概念

比較演算子は、式の左右の値を比較させ「大きい」「小さい」「等しい」などの判断をさせるものです。

したがって、「正しい」や「誤っている」という答えが返ってきます。

コンピュータでは、

正しい場合には	**真（しん）**	**“0以外の値”**
誤っている場合には	**偽（ぎ）**	**“0”**

が返ってきます。

比較演算子

=	AとBは等しい（[例] A=B）
<	AはBより小さい（[例] A<B）
>	AはBよりも大きい（[例] A>B）
<=	AはBよりも小さいか等しい（A<=B）
>=	AはBよりも大きいか等しい（A>=B）
<>	AとBは等しくない（[例] A<>B）

2　比較演算子の使い方

比較演算子は、「正しい」「正しくない」という演算結果を数値として返すと述べましたが、その結果は通常、見ることはできません。たいていは、次章で解説するIF文などの条件分岐で、その分岐先を決定するために使われます。

ここでは、結果をそのまま表示する方法を使って、比較演算の結果を見てみましょう。

●ソースリストPRG-NO21.BAS

```
CLS
A = 3
B = 2
PRINT "A ="; A; "B ="; B
PRINT
PRINT "A < B :"; A < B
PRINT "A > B :"; A > B
PRINT "A <= B :"; A <= B
PRINT "A >= B :"; A >= B
PRINT "A <> B :"; A <> B
```

●インタプリタによる実行結果

```
A = 3 B = 2

A < B : 0      ←――Aは Bより小さくないので偽（0）となる
A > B : -1     ←――Aは Bよりも大きいので真（0以外の数値）となる
A <= B : 0     ←――Aは Bよりも等しくもなく大きくもないので偽（0）となる
A >= B : -1    ←――Aは Bよりも等しいもしくは大きいので真（0以外の数値）となる
A <> B : -1    ←――Aと Bは等しくないので真（0以外の数値）となる
```

ここでは ＝ を使った比較演算を行なっていません。＝ を使うと、変数のところで行なった代入を行なってしまい、正しい結果が得られません。

＝ 、つまり「等しい」場合については次章で扱います。

7・3

論理演算

複合した比較の結果を求める

論理演算は、比較演算を組み合せたように使います。AとBを比較し、その後さらにもう一つ比較を行ない、それらを複合した結果を求めます。ここでは、論理演算の全体像をつかんでおきましょう。実例は、次章で取り上げます。

論理演算子

NOT（否定）　A以外
　[例] NOT A

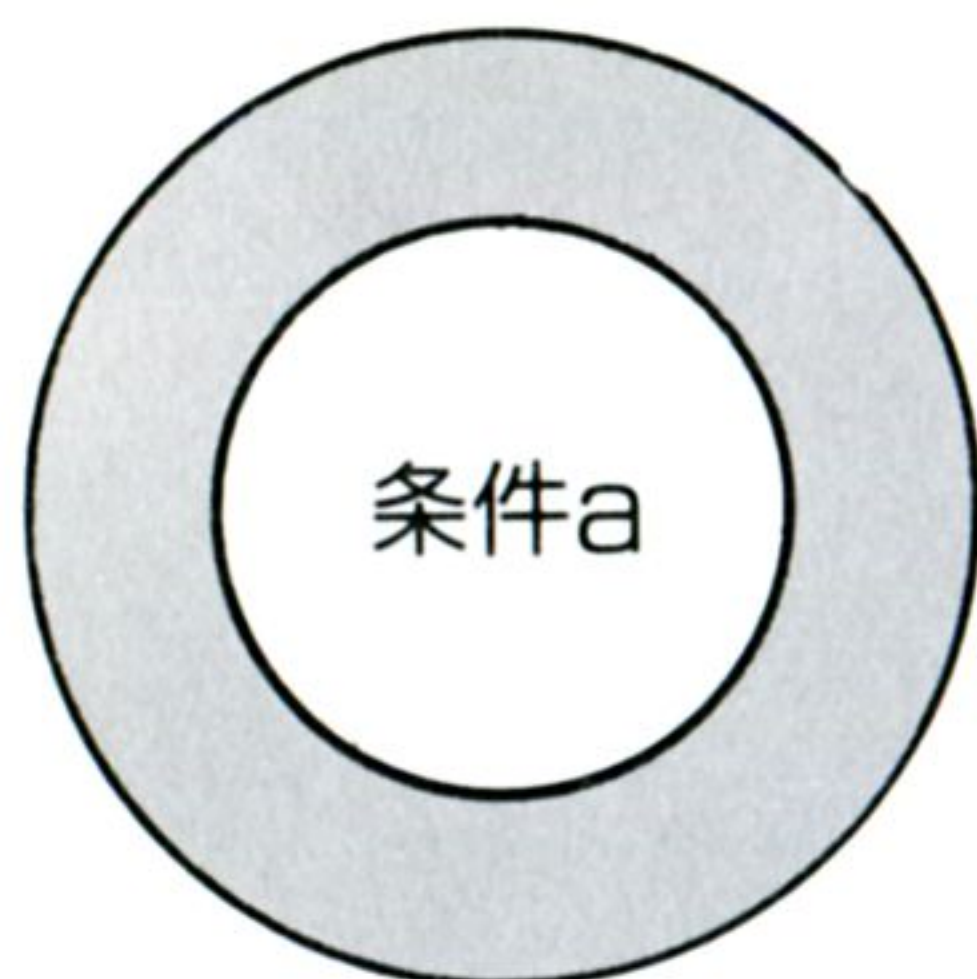
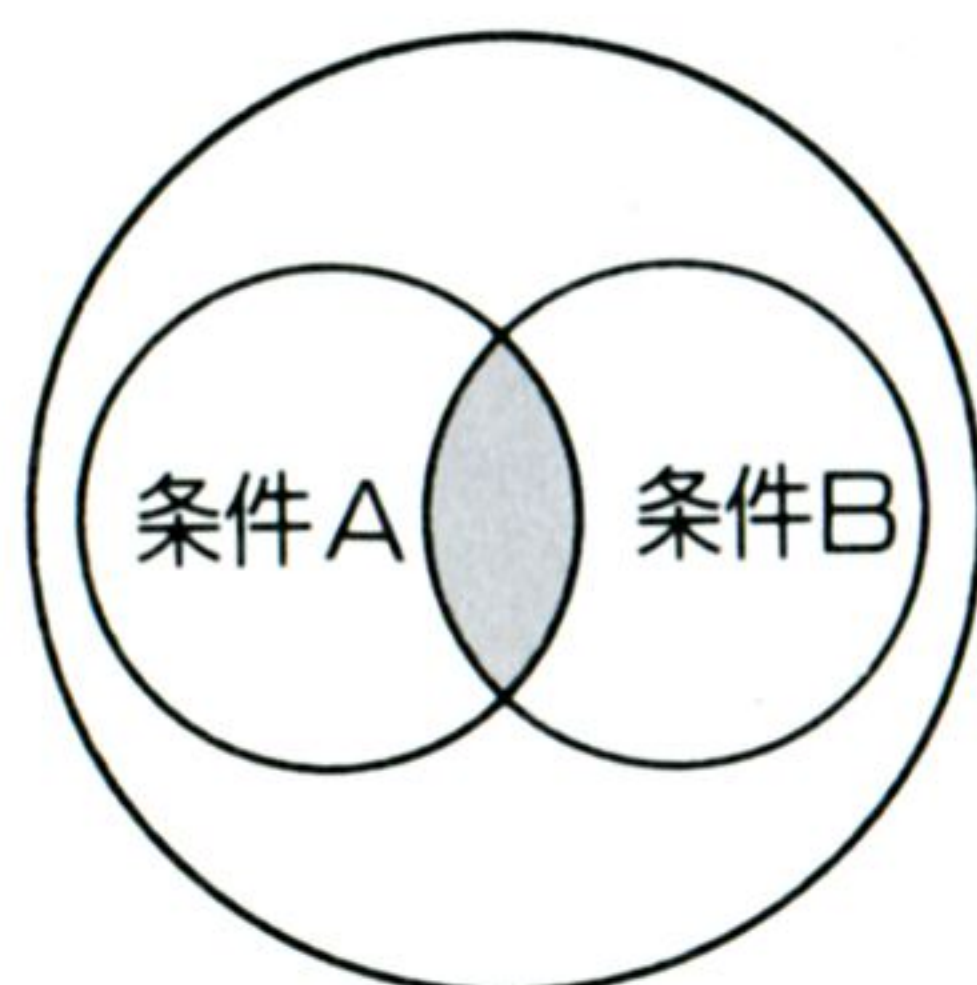
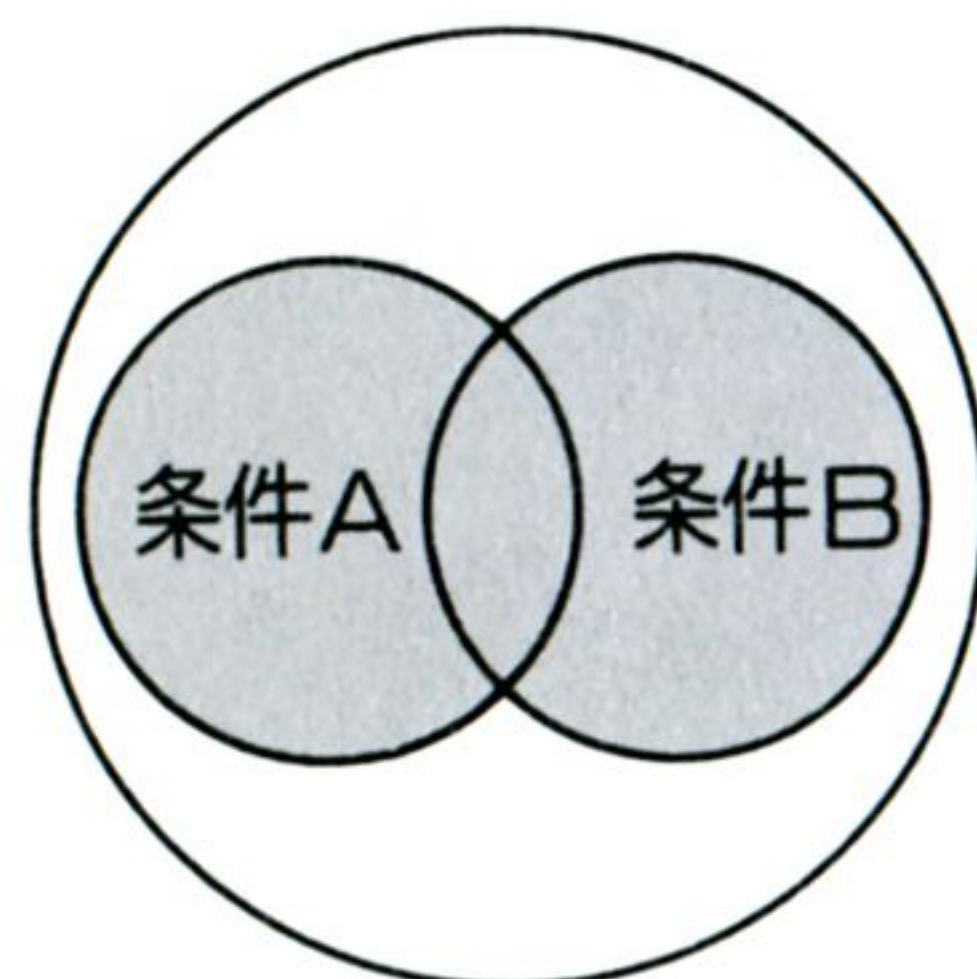
AND（論理積）　Aの値と一致し、かつBの値と一致する
　[例] A AND B

OR（論理和）　Aの値もしくはBの値と一致する
　[例] A OR B

このように「集合」を行なうのが、論理演算です。
論理演算の場合にも比較演算同様、答えは数値で返ります。

「偽」の場合には“0”
「真」の場合には“0”以外の値

が返ります。
これらの使い方については、先ほども述べた通り次の章で解説します。

●論理演算の概念

否定：NOT A
条件 A以外の場合のみ真となる。

論理積：A AND B
条件Aを満たしていて、かつ条件Bを満たしている場合だけ真となる。

論理和：A OR B
条件Aでも条件Bでもどちらでも真となる。

●注意　斜線の部分が条件を満たしている、つまり真

第8章

条件分岐

これまで、BASICの基本的な事柄を学んできました。今まで学んできたことだけでもちょっとしたプログラムを書くことができます。しかし

繰り返し

条件を判断して分岐する

など、コンピュータがもっとも得意とすることは、これだけではできません。これらのことをコンピュータにやらせてこそ、コンピュータのプログラミングというものなのです。

このような「繰り返し」「条件分岐」のことを、プログラミングの制御と言いますが、ここからはこの「制御」について学んでいきます。

8・1

条件分岐

二者択一と複数択一

コンピュータのプログラミングにおいて、条件分岐はたいへん重要な基本的テクニックの1つです。

何かのプログラムを書く場合、ある条件によって、YESならばアレをさせたい、NOならばコッチのことをさせたいというケースは、まま必要とされます。

また、いくつかの項目の中から、ある1つの条件に合うものを選択させたいなどというケースもよく出てきます。

ちょっとでもプログラミングをかじったことのある人や、MS-DOSでメニュー選択のできるバッチファイルを作ったことのある人ならすぐに、概念は把握できるはずです。

そして「言語ってヤッパリ難しいや」となってしまうのもこの辺りからなのです。初めての人も、そんな経験のある人も、ちょっと気を引きしめてとりかかって下さい。

BASICの条件分岐は、次の2つに分かれます。

①IF文

二者択一を行なう。2つある条件のうち、どちらかを満たしている方に分岐させる。

②SELECT文

複数択一を行なう。3つ以上の条件のうち1つを満たしている方に分岐させる。

8・2
IF文

IF...はプログラミング言語の基本中の基本

1　IF文の記述方法

IF文は、次のように記述します。

◉IF文の記述

```
IF 条件 THEN
      実行文1;      ── A
      実行文2;
    ELSE
      実行文1       ── B
      実行文2;
    END IF
```

■「条件」を満たしている場合

Aに書かれた実行文を実行し、ELSE以下の文を飛ばして、END IF以降に制御を移します。

■「条件」を満たしていない場合

Aに書かれた実行文を飛ばします。Bに書かれた実行文を実行し、END IF以降に制御を移します。

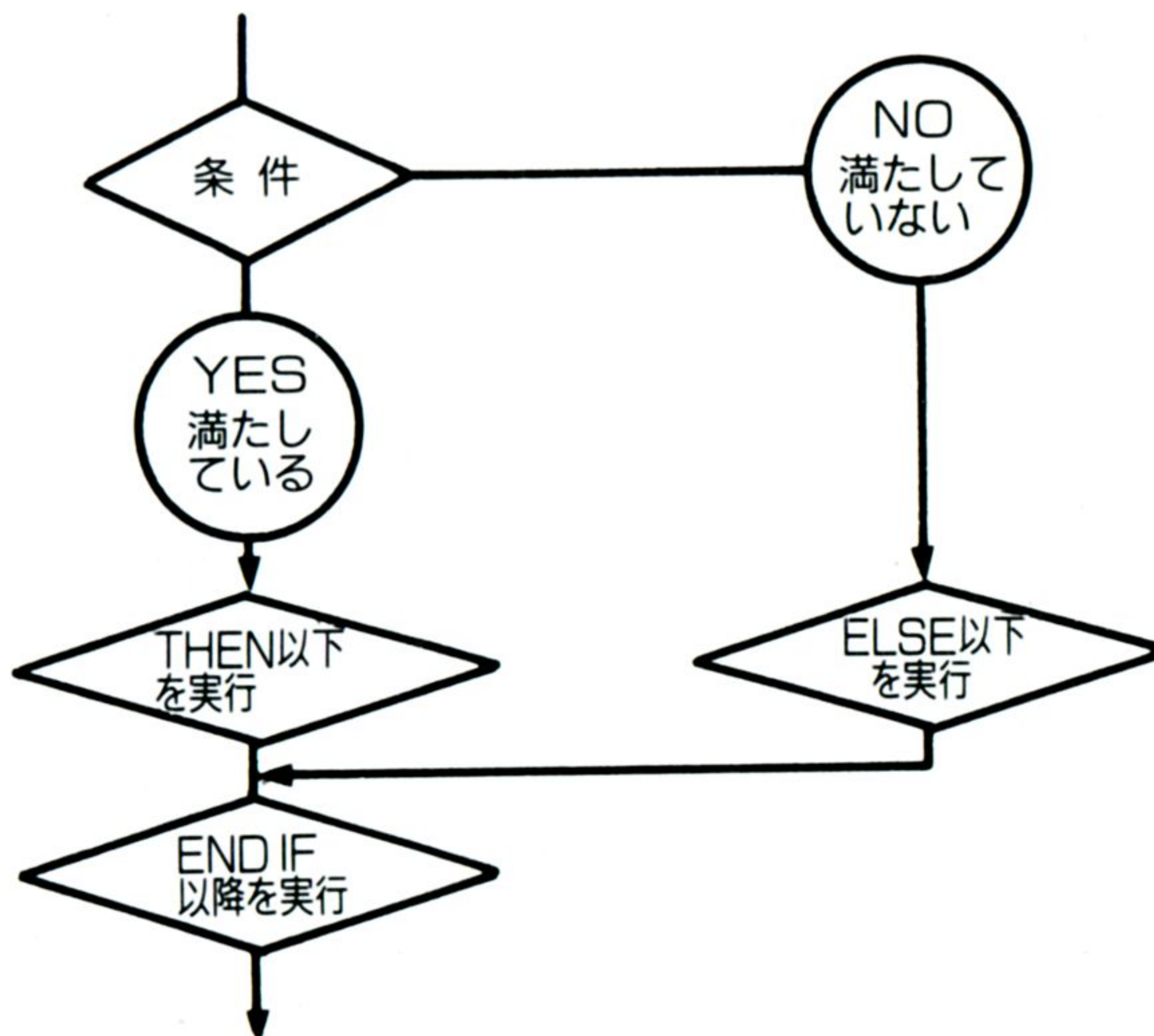
■END IF

このEND IFとは、「IF文の終りはここである」という印なので、IF文を使うときにはかならず必要です。

■条件

「条件」には、通常は第７章で学んだ比較演算や論理演算を入れます。
条件の判断として「等しいか／等しくないか」や「大きいか／大きくないか」などのような比較を用いるのです。

以上のことを、フローチャートに表現すると、次のようになります。

◉IF文のフローチャート

2　IF文の使い方

では、実際にIF文を作ってみましょう。

●ソースリストPRG-NO22.BAS

```
CLS
INPUT "Please DATA1 : ", A  ←―変数Aに数値を代入
INPUT "Please DATA2 : ", B  ←―変数Bに数値を代入
      ┌――――――――――――これが条件
IF [A = B] THEN
   [PRINT "A = B"]←――①
ELSE
   [PRINT "A <> B"]←――②
END IF
```

ここでは「条件」が、「A = B」です。つまり、変数AとBが等しい場合には条件を満たしているので①を実行し、AとBが等しくなければ②を実行します。

次に、END IF以降に制御を移しますが、ここではEND IF以降にステートメントがないのでそのまま終了となります。

●インタプリタによる実行結果1

```
Please DATA1 : [3]
Please DATA2 : [3]
A = B
 ↑―――――――変数Aと変数Bの内容が等しいので「=」で表示
```

これは、最初に入力した値と次に入力した値が等しいため“A = B”が表示されました。つまり、“IF A = B...”という**条件を満たした**のです。

●インタプリタによる実行結果2

```
Please DATA1 : 1
Please DATA2 : 2
A <> B
```

変数Aと変数Bの内容が等しくないので「<>」で表示

今度は、最初に入力した値と次に入力した値が等しくありません。したがって“IF A = B...”という**条件を満たしていない**ので“A <> B”を表示しました。

これで、IF文の基本を把握できたと思います。

IF...THENはプログラミング言語の基本中の基本だということを忘れないでください。

次のプログラムは、内部に設定した値と入力した値が等しいか検査し、等しくない場合にはその旨のメッセージを出力して終了します。

つまり、パスワードなどをチェックする、もっとも基本的なプログラムです。

●ソースリストPRG-NO23.BAS

```
CLS
ORG$ = "ABCDE"
INPUT "Pasword : ", PASWORD$

IF ORG$ = PASWORD$ THEN
      PRINT "OK!!!"
ELSE
      BEEP ←ブザーをならすテクニック
      PRINT "You are not menber !!!"

END IF
```

ここの空白は動作には関係ない

IF文では、数値だけでなく文字なども条件の判断として扱えます。ここではその例を行ないました。

●インタプリタによる実行結果1

```
Pasword : ADE14
You are not menber !!!  ←オリジナルのパスワードと違うの
                          でエラーメッセージ出力
```

●インタプリタによる実行結果2

```
Pasword : abcde
You are not menber !!!  ←オリジナルのパスワードとスペルは合ってい
                          るが大文字でないのでエラーメッセージ
```

さて、今度の実行結果では、スペルはオリジナルのパスワードと合っているのに、「等しくない」とされてしまいましたね。

●インタプリタによる実行結果3

```
Pasword : ABCDE
OK!!!  ←オリジナルのパスワードと一致したので"OK!!!"の
         メッセージを得ることができた
```

そうです。Quick BASICは「大文字と小文字を区別する」言語だったのです。Quick BASICに限らず、コンピュータ言語ではほとんど大文字と小文字を区別します。

MS-DOSに慣れてしまうと、ちょっと戸惑い勝ちですが、プログラミングを行なうには、こういったことにも気を付けましょう。

8・3

SELECT文

変数や式が条件と一致すれば、実行文を実行

1 SELECT文の記述方法

SELECT文は、条件に指定された変数や式を参照し、その条件とそれぞれの要素が一致するかをどうかを判断して、それらの要素以降の実行文を実行します。

SELECT文は、次のように記述します。

◉SELECT文の記述

```
SELECT CASE 条件
        CASE        要素1
                    実行文1;
                    実行文2;
                      ⋮
        CASE        要素2
                    実行文1;
                    実行文2;
                      ⋮
        CASE ELSE
                    実行文1;
                    実行文2;
                      ⋮
END SELECT
```

■条件で示した値と要素１で示した値を評価

まず最初に、条件で示した値と要素１で示した値を評価します。もし、条件で示した値を要素１が満たしている場合には、要素１以下の実行文を実行し、“CASE”が現れた時点で、制御を“END SELECT”以降に移します。

■要素１が条件を満たしていなければ要素２を評価

また、条件で示した値と要素１で示した値が満たされていない場合には、次の要素２と条件を評価します。

そして、もし条件で示した値を要素２で示した値が満たしていれば要素２以下の実行文を実行し、“CASE”が現れた時点で、制御を⑤の“END SELECT”以降に移します。

■CASE 要素...

このように、“CASE 要素...”とすれば、それぞれの要素を満たす実行文だけを固めて、メニューなどの分岐に使えます。

■全ての要素を満たさない場合、ELSE CASE以降を実行

また、全ての要素を満たさないような値が入力された場合には、“ELSE CASE”以降に実行文を書き、それ以降を要素を満たさなかった場合に対応させます。

これを、フローチャートにすると次のようになります。

●SELECT文のフローチャート

条件

要素1

YES
満たしている

実行文

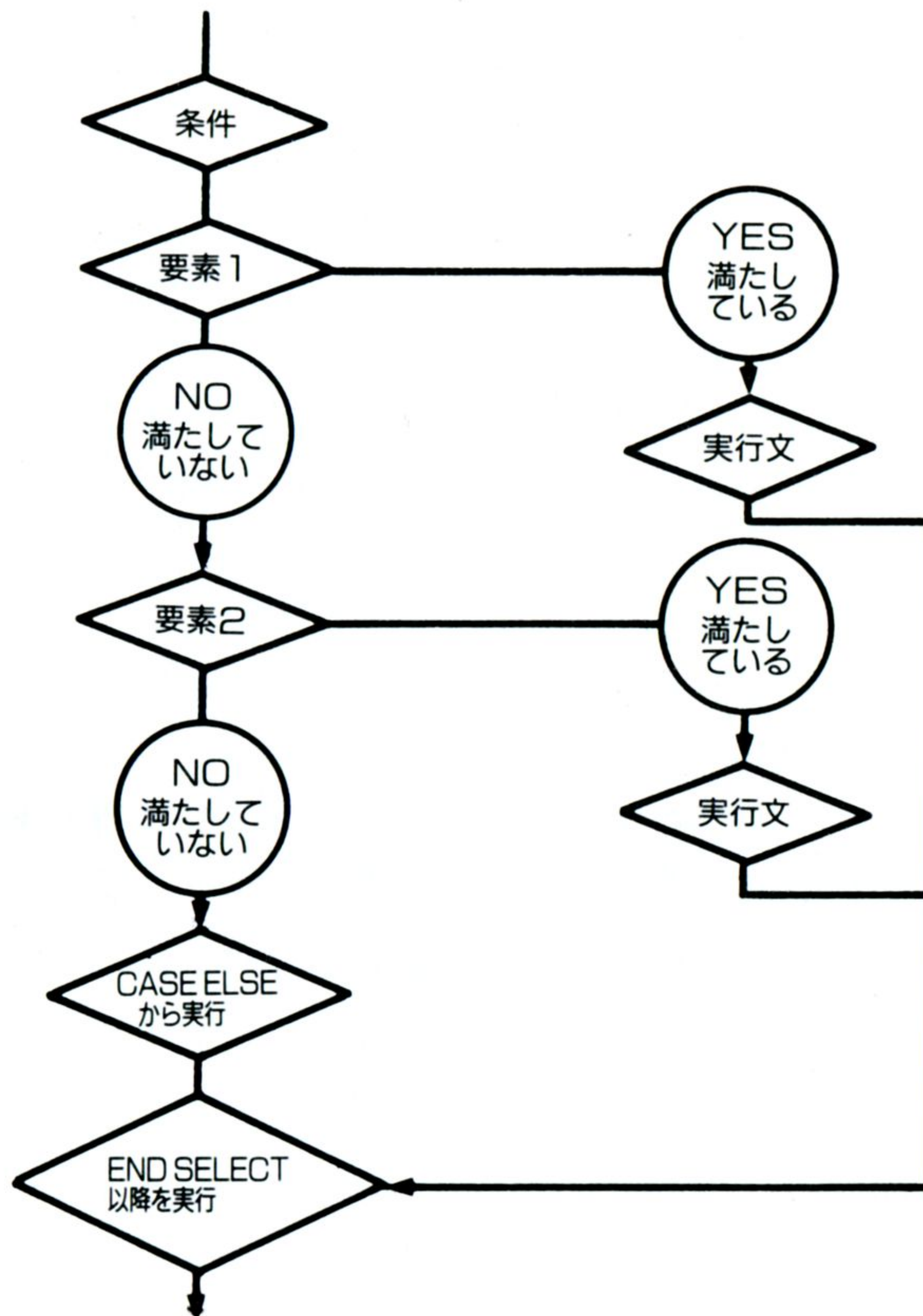
NO
満たしていない

要素2

YES
満たしている

実行文

NO
満たしていない

CASE ELSE
から実行

END SELECT
以降を実行

2　SELECT文の使い方

実際にプログラミングを行なってみましょう。

◉ソースリストPRG-NO24.BAS

```
CLS
PRINT "Do you use machine ?"
INPUT "F2 = 1,VM = 2, VX = 3, Other machine = ELSE : ", M

SELECT CASE M
    CASE 1
        CPU$ = "8086"
    CASE 2
        CPU$ = "V30"
    CASE 3
        CPU$ = "80286"
    CASE ELSE
        CPU$ = "80386"
END SELECT

PRINT "Your machine <CPU> is "; CPU$
```

このサンプルプログラムは、PC-9801のCPUが何であるかを答えさせるためのプログラムです。

まず最初に、INPUT文を使ってマシンのナンバーを変数Mに格納します。その次にSELECT文で変数Mで格納した数値を判断、その後その数値に対応したCASE文以降を実行します。

CASE文では、文字型変数CPU$にCPUのタイプを格納し、最終的にPRINT文で変数CPU＄の内容を表示します。

◉インタプリタによる実行結果1

```
Do you use machine ?
F2 = 1, VM = 2, VX = 3, Other machine = ELSE : 1
Your machine <CPU> is 8086
```

●インタプリタによる実行結果2

```
Do you use machine ?
F2 = 1, VM = 2, VX = 3, Other machine = ELSE : 5
Your machine <CPU> is 80386
```

1～3までのマシンは（F2, VM, VX）ちゃんと判断できますが、それ以外の数値は全て80386のCPUを使っていると判断されてしまいます。

9801F2の後期のマシンでは8086IIを搭載していますし、9801UV2などでもV30を搭載しているので、一概にこの通りとはいえませんが、ここではSELECT文の動作を確認するためのサンプルなので、おおまかに分類させていただきました。

次にSELECT文で範囲指定をして、それぞれの分岐をさせてみましょう。

●ソースリストPRG-NO25.BAS

```
CLS
INPUT "When is "; MONTH

SELECT CASE MONTH
    CASE 1 TO 2, 12 ←――――― ②飛石となる場合はカンマで区切る
        SEASON$ = "Winter"
    CASE 3 TO 5
        SEASON$ = "Spring"
    CASE 6 TO 8   ←――――― ①連続している要素を範囲確定
        SEASON$ = "Summer"
    CASE 9 TO 11
        SEASON$ = "Autumn"
    CASE ELSE
        SEASON$ = "nothing !!!"
END SELECT

PRINT MONTH; "is "; SEASON$
```

SELECT文は「～以上、～以下」を要素とする使い方ができます。

「○○～○○」までといった範囲を要素に指定する場合には、“TO”を使って要素を区切ります。

この場合には、要素が連続していなければなりません。

サンプルリストでは、①が該当します。

また、要素が連続しておらず、飛石のようになっている場合には、カンマで区切って指定します。サンプルリストでは、②が該当します。

このように、SELECT文は様々な対応があるので、実際のプログラミングでも欠かせない存在となるのはいうまでもありません。

f･5キーを押して、実行してみましょう。

◉インタプリタによる実行結果1

```
When is ? 1
 1 is Winter
```

◉インタプリタによる実行結果2

```
When is ? 14
 14 is nothing !!!
```

第9章

繰り返し

条件分岐の次は、繰り返し処理を学びます。
「条件分岐」もコンピュータ言語の要となりますが、「繰り返し」もまたコンピュータ言語では、必須の収得科目です。

コンピュータは、「決った仕事を繰り返す」ことが大の得意ですので、繰り返し処理を収得しなければ、コンピュータ言語は使えないといったことにもなってしまいます。

ここで、繰り返し処理の基本をマスターしておきましょう。

9・1

繰り返し

FOR文、WHILE文、DO文

BASICの繰り返しには、次の3つがあります。

①FOR文
②WHILE文
③DO文

■FOR文

FOR文は、指定回数だけ、ステートメントを繰り返し実行します。つまり、あらかじめ繰り返しを行なう回数が明らかな場合（カウントをとるなど）に使います。

■WHILE文

WHILE文は、条件を満たさなくなるまで指定のステートメントを繰り返して実行します。繰り返し処理をする回数が不明な場合や、条件によって繰り返し処理を終了させる場合に使用します。

■DO文

DO文は、基本的にはWHILE文と同じ動作をします。

ただし、DO文の場合にはステートメントの最後(LOOPまでくると)に、もう一度、条件の評価ができます。

9・2
FOR文
繰り返し処理（ループ構造）

1　FOR文の記述方法

FOR文は、BASICプログラムの基本中の基本といわれるほど、よく使われる繰り返し処理（ループ構造）です。次のように記述します。

◉FOR文の記述

FOR 初期値 **TO** 最終値 **STEP** 増量
　　ステートメント
　　ステートメント
　　　：
NEXT 変数名

■初期値のセット

まず最初に、変数を使った初期値をセットします。次に、BASIC内部のカウンタに初期値をセットします。これで第一段階は終了します。

■「BASIC内部のカウンタよりも終了値が大きいか」を判断

次に、変数名を明記したNEXT（通常は）に出会うまでステートメントを実行し、NEXTまで制御が移されたときに「BASIC内部のカウンタよりも終了値が大きいか」を判断します。

この時、終了値にセットした値よりもカウンタの方が大きければ、カウンタの値から１を引いて、再度FOR文のステートメントに制御を移します。

■カウンタの内容が終了値よりも大きければ終了

　このようにして繰り返し処理を実行した後、NEXTでカウンタと終了値を調べたとき、カウンタの内容が終了値よりも大きければ繰り返し処理は終了となります。

■STEPオプション

　また、FOR文には「STEPオプション」があります。

　STEPをFOR文の最後に付け加え、増量に数値を書き込むと、書き込んだ数値分、一度にカウントする量を増やします。

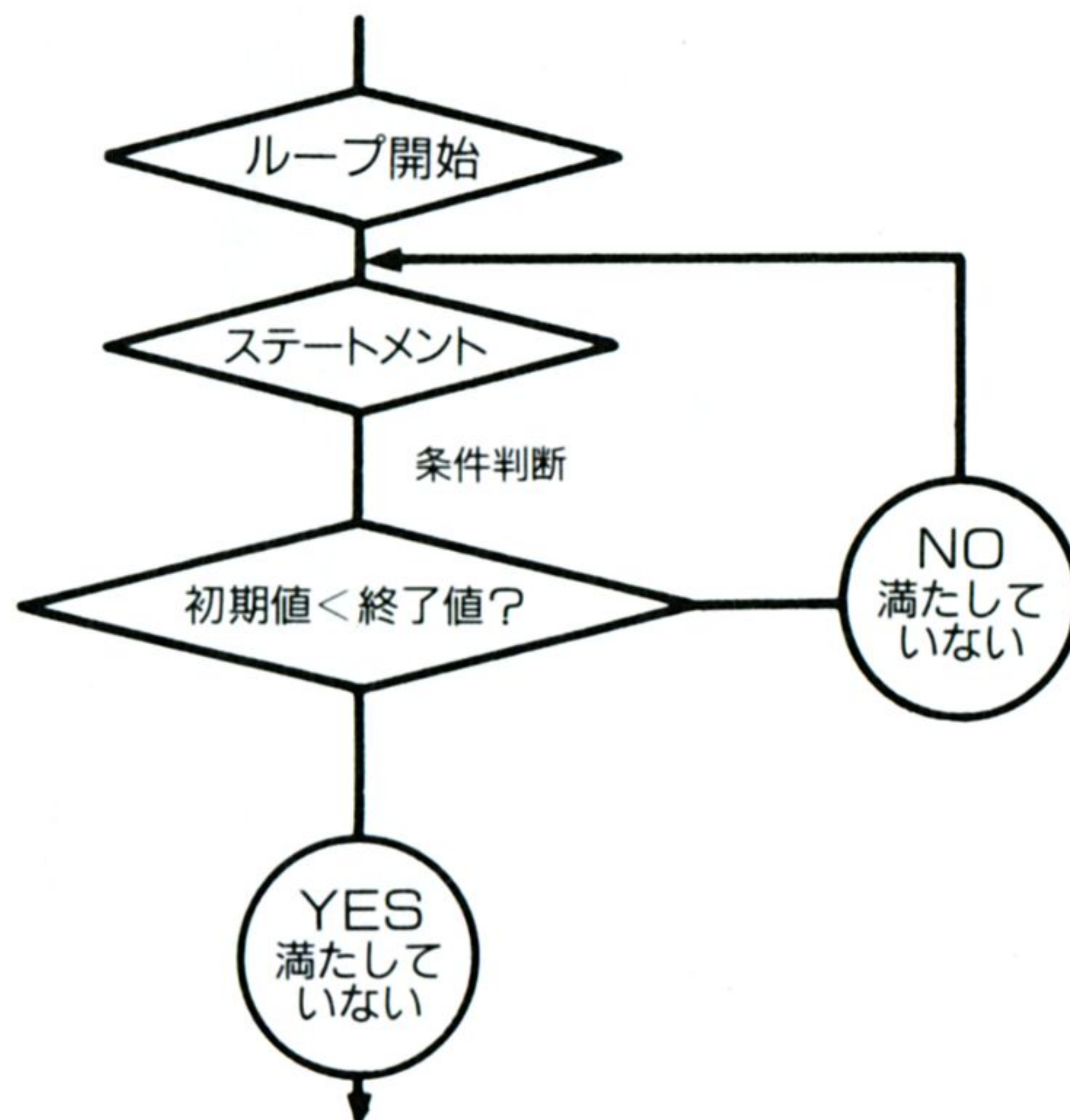
　例えば、

```
FOR X = 1 TO 100 STEP 2 ← これが増量分
```

とした場合には、一度にカウントする値が2づつになります。

　FOR文のフローチャートは、次のようになります。

●FOR文のフローチャート

ループ開始
ステートメント
条件判断
初期値＜終了値？
NO
満たして
いない
YES
満たして
いない

2　FOR文の使い方

ループ構造を知るために、簡単なカウントプログラムを作って、FOR分の動作を確かめてみましょう。

●ソースリストPRG-NO26.BAS

カウントをとる初期値
「X＝1」なので初期値は“1”となる

```
CLS
PRINT "Count program <1-1000>"
FOR COUNT = 1 TO 1000
    LOCATE 1, 23
    PRINT COUNT
NEXT COUNT
```

カウントの終了値
「1000」なので“1000”になるまでステートメントを実行
カーソルの位置を固定
変数Xの内容を表示

初期値を１とし、終了値を1000としているので、このループ構造のステートメントは1000回実行されることになります。

また、ステートメントの中に「LOCATE 1，23」というのがありますが、これを入れないとPRINT文で次の行にカーソルが移動するため、カウンタとしての機能が働かなくなるので注意してください。

実行してみましょう。

●インタプリタによる実行結果

```
Count program <1-1000> 1000
```

さて、もう1つプログラムを紹介しましょう。

●ソースリストPRG-NO27.BAS

```
CLS
PRINT "Calc program"

INPUT "Please input data : ", CALC
IF CALC <= 0 OR CALC >= 10 THEN
    PRINT "Over the limits !!!"          ←①
    END ←──── プログラムを強制終了させる
END IF

FOR COUNT = 1 TO 9
    PRINT CALC; "*"; COUNT; "="; CALC * COUNT   ←②
NEXT COUNT
```

●インタプリタによる実行結果1

```
Calc program
Please input data : 8
  8 * 1 = 8
  8 * 2 = 16
  8 * 3 = 24
  8 * 4 = 32
  8 * 5 = 40
  8 * 6 = 48
  8 * 7 = 56
  8 * 8 = 64
  8 * 9 = 72
```

●インタプリタによる実行結果2

```
Calc program
Please input data : 0 ┐
Over the limits !!!   ┘── 範囲を超えた入力を行った
```

PRG-NO27.BASのポイント

①IF文

①の部分で入力したキーのエラーチェックを行なっています。

エラーチェックの対象は、変数CALCです。変数CALCに、もし“0”以下（CALC <= 0）もしくは“10”以上（CALC >= 10）の値が入力された場合には、IF文のステートメントを実行します。

つまり、「Over the limits !!!」と表示し、ENDステートメントを実行して強制的にプログラムを終了させます。

ENDステートメントについては、ここで初めて出てきましたが、これからのプログラムにはかならず入れてください。

今までのように、ごく簡単なプログラムでは、ENDステートメントなどなくとも終了位置にくれば、Quick BASICが勝手に終了させますが、サブルーチンなどを使うようになるとENDステートメントが必要となります。

②FOR文

FOR文の本体です。

まず、PRINT文で変数CALCの内容を表示させています。

次に、×を示す「＊」をそのまま表示しています。そして、変数COUNTの内容をそのまま表示、「＝」、最後に変数CALCと変数COUNTの計算結果をそのまま表示しています。

9・3

WHILE文

条件を満たしている間は、ステートメントを実行

1 WHILE文の記述方法

WHILE文は、条件を満たしている間は、ステートメントを実行し続けます。次のように記述します。

●WHILE文の記述

```
WHILE 条件
        ステートメント
        ステートメント
              ⋮
WEND
```

■ステートメントを一度も実行しないループ構造を作れる

WHILE文はステートメントを実行する前に、条件を判断しますので、ステートメントを一度も実行しないループ構造を作り出すこともできます。

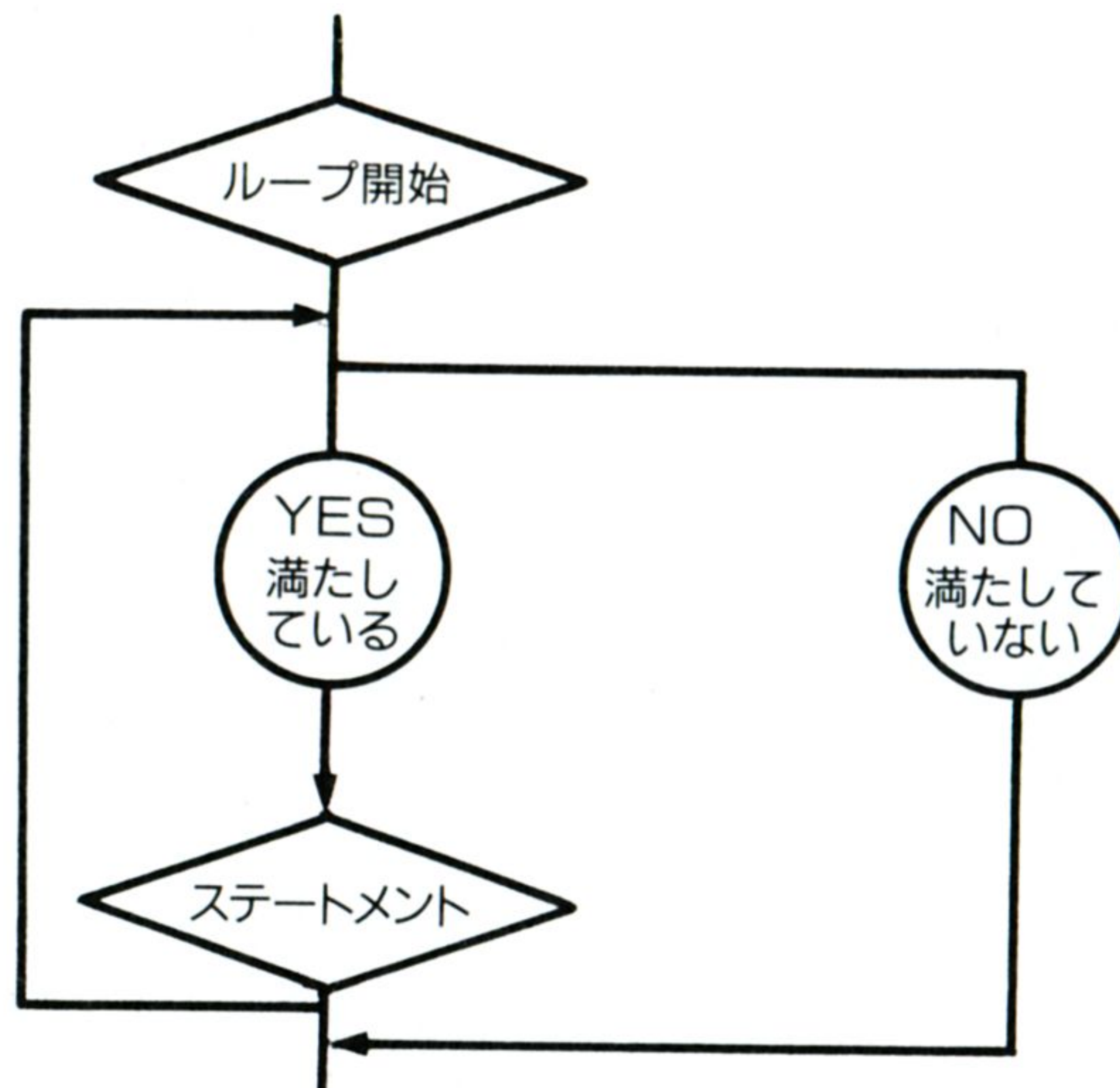
■キー入力判断などに使える

FOR文がカウンターなどに使うことを目的とするならば、WHILE文は、キー入力の判断などによく使われます。

これをフローチャートにすると、次のようになります。

ループ開始

YES
満たしている

NO
満たしていない

ステートメント

2　WHILE文の使い方

WHILE文を使って、プログラミングしてみましょう。

●ソースリストPRG-NO28.BAS

```
CLS
PRINT "Inputkey check program"
INPUT "Please input key [Y/N] : ", CHECK$

WHILE CHECK$ <> "Y" AND CHECK$ <> "N"  ←①
    INPUT "Please input key [Y/N] : ", CHECK$  ←②
WEND  ←③WHILE文の終わりを示すステートメント
PRINT "OK !"
```

PRG-NO28.BASのポイント

①WHILE CHECK$ <> "Y" AND CHECK$ <> "N"

WHILE文は、「ループに入る前に条件を判断する」と述べましたが、このリストを見ても分かると思いますが、ステートメントを行なう前に条件を書き込んでいます。

複数の条件が入っているので、それぞれを分解してみましょう。

CHECK$ <> "Y"　CHECK$ <> "N"

両者共に、変数CHECK$の内容を調べています。演算子に、**以外**を示す"＜＞"を使って、関係演算を行なっています。

つまり「変数CHECK$の内容が"Y"または、"N"以外の場合には」となります。

AND

この"AND"は、論理演算の"AND"です。"AND"を使った場合には、「〜であり、かつ〜」となるので、ここでは2つの条件を判断させていることになりますね。

これを日本語で表現すると「変数CHECK$が"Y"以外であり、かつ変数CHECK$が"N"以外の場合にはWHILE文のステートメントを行ないなさい。」となります。

つまり、"Y"もしくは"N"を入力しない限り、条件を満たしてしまうのでWHILE文のステートメントを実行してしまいます。

インタプリタで実行してみましょう。

●インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : Z
Please input key [Y/N] : 1
Please input key [Y/N] : y
Please input key [Y/N] : Y
OK !
```

最初に“Z”を、次に“1”を、そして次に“y”を入力していますが、全て再入力になっています。これらの入力は、WHILE文の条件を満たしています。

そして、最後に“Y”の入力を行なって、初めて条件を満たさなくなりました。したがって、次からのステートメントは実行されず、WENDステートメントまで制御が移されるという仕組みです。

3　無限ループ

PRG-NO28ではキーを入力した後、⏎キーを押さなくてはなりませんでした。

⏎キーを押さずに、1文字入力した時点で入力したキーを判断する方法はないものでしょうか。

これには、INKEY$という関数を使うます。

さっそく、INKEY$を使ったプログラムを作ってみましょう。

●ソースリストPRG-NO29.BAS

```
CLS
PRINT "Inputkey check program"

WHILE 1
    LOCATE 2, 1 ←―― 表示位置固定
    PRINT "Please input key [Y/N] : "; K$
    K$ = INKEY$ ←――――――――― キー入力の検査
    IF K$ = "Y" OR K$ = "N" THEN ←―― 入力されたキーの検査
        PRINT "OK !"
        END
    END IF
WEND
```

INKEY$関数は、INPUT文などと違い、キー入力があろうとなかろうと次のステートメントに制御を移します。

この機能を使って、リターンキーを押さなくても、入力したキーが何であるかを判断させてみました。

ただしINKEY$関数では、先ほども述べた通りキー入力のあるなしに関わらず次のステートメントに制御を移しますので、PRINT文などがある場合には次々と表示してしまいます。そこで、LOCATE文で表示を固定させています。

●インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : N
OK !
```

「無限ループ」のテクニック

ここでは「無限ループ」というテクニックを使っています。
「無限ループ」とは、終りがないループ構造を故意に作成することです。

本来、WHILE...の後には**条件**がくるはずですが、ここでは“1”が入っています。これは、条件の戻り値を応用した例です。
「条件」でも、判断された結果は、必ず数値で返されたのを覚えていますか？

WHILE文では「条件が満たしている間」(条件が真[0以外の場合])は、ループを回り続けます。

つまり、条件に直接“0”以外の数値を書き込めば無限ループになります。

プログラミングが初めてという人には、ちょっと難しかったかもしれませんが「無限ループ」は、重要なテクニックの1つなので、慣れながら身に付けていってください。

9・4

DO文

不特定な条件の繰り返し処理

1　DO...LOOP文の記述方法

DO...LOOP文は、基本的にWHILE文と似ています。不特定な条件を指定し、繰り返し処理を行ないます。次のように記述します。

◉DO...LOOP文の記述

```
DO 条件
    ステートメント
    ステートメント
        ⋮
LOOP
```

■満たす／満たさないの条件を指定できる

WHILE文では条件を満たさなくなった時点で繰り返し処理を中断しますが、DO...LOOP文では、条件を「満たす」「満たさない」の条件を指定することができます。

また、WHILE文の条件の判断は、WHILE文の先頭で固定していましたが、DO...LOOP文では、先頭／末尾どちらでも条件の判断ができます。

ここに示した記述方法は、もっとも基本的なDO...LOOP文の構造ですが、このほかにもいくつかの構造を持っています。

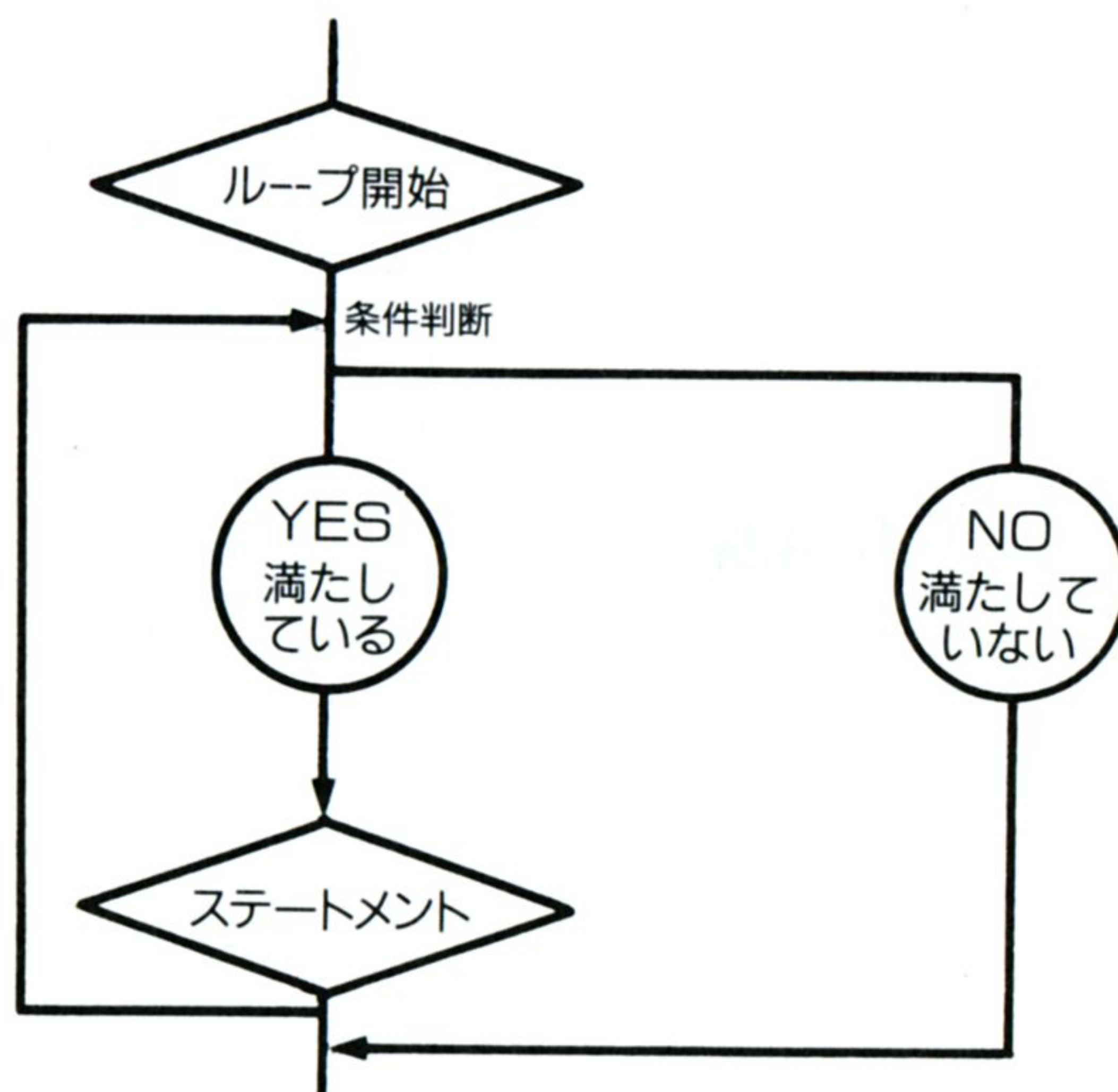
■EXIT LOOP文

　また、EXIT LOOP文を使えば、強引にDO...LOOP文から抜けることができます。

　WHILE文では、条件の判断などが特定していたため頭をひねらないとならないような処理でも、DO...LOOP文を使えば、柔軟に対応できるわけです。

　これをフローチャートにすると、次のようになります。

◉DO...LOOP文のフローチャート

ル--プ開始
条件判断
YES
満たしている
NO
満たしていない
ステートメント

2　先頭で条件判断を行なう

最初に、DO...LOOP文の先頭で条件を判断する方法を見てみましょう。

◉ソースリストPRG-NO30.BAS

```
CLS
PRINT "Inputkey check program"
INPUT "Please input key [Y/N] : ", CHECK$
    ↓―――――――――― 満たさなくなるまで実行
DO WHILE CHECK$ <> "Y" AND CHECK$ <> "N"
    INPUT "Please input key [Y/N] : ", CHECK$
LOOP
PRINT "OK !"
```

◉インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : Z
Please input key [Y/N] : Y
OK !
```

この例は、WHILE文のところで行なったリストをそのままDO...LOOP文で行なったものです。

WHILE文と同じように条件を先頭で判断させ、かつ"満たさなくなるまで"とするには"DO WHILE..."とします。DOの次にWHILEを付ければ全くWHILE文と同じになります。

では、"満たすまで"とするにはどうしたらいいのでしょうか。

●ソースリストPRG-NO31.BAS

```
CLS                  ┌──────── 満たすまでにするには
PRINT "Inputkey check program"
INPUT "Please input key [Y/N] : ", CHECK$
DO UNTIL CHECK$ <> "Y" AND CHECK$ <> "N"
    INPUT "Please input key [Y/N] : ", CHECK$
LOOP
PRINT "OK !"
```

条件を満たすまでにするには、先ほど“WHILE”としたところを“UNTIL”に変更するだけです。

実行してみましょう。

●インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : Y
Please input key [Y/N] : N
Please input key [Y/N] : A
OK !
```

“Y”と“N”だけしか、入力を受け付けられていません。つまり、「条件を満たしている」ので、ループから抜けられないことになります。

このようにDO...LOOP文では、条件判断形式の変更ができます。

3　条件判断を末尾で行なう

今度は、条件をループ構造の最後で判断させてみましょう。

条件を末尾で行なった方が、スッキリするプログラムはいくらでもあります。

◉ソースリストPRG-NO32.BAS

```
CLS
PRINT "Inputkey check program"

DO
    INPUT "Please input key [Y/N] : ", CHECK$
LOOP WHILE CHECK$ <> "Y" AND CHECK$ <> "N"
PRINT "OK !"
```

条件を末尾で判断しているので、今までDO...LOOP文に入る前に使用していた“INPUT ...”が必要なくなりました。

また、先ほどと同じように末尾で条件を判断する場合でも、“LOOP WHILE”とすると“条件を満たさない場合”となるので注意してください。

実行してみましょう。

◉インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : D
Please input key [Y/N] : Y
OK !
```

これと同様に、“LOOP WHILE” を “LOOP UNTIL” に変更すれば、“条件を満たす場合” に変更できることはいうまでもありせん。

●ソースリストPRG-NO33.BAS

```
CLS
PRINT "Inputkey check program"

DO
    INPUT "Please input key [Y/N] : ", CHECK$
LOOP UNTIL CHECK$ = "Y" OR CHECK$ = "N"
PRINT "OK !"
```

“Y” と “N” を入力するとループを抜けるように作成しました。

●インタプリタによる実行結果

```
Inputkey check program
Please input key [Y/N] : D
Please input key [Y/N] : Y
OK !
```

PRG-NO33のポイント

注意しなくてはならないのは、条件の判断方法が変わった点です。「満たすまで」となっていますから、“Y” と “N” だけを満たすとしなくてはなりません。

<> (以外) から= (等しい) に変更しました。また、AND (かつ) のままではループを抜けられなくなりますので、ここもOR(もしくは)に変更しました。

```
                                    ┌ANDからORへ変更
LOOP UNTIL CHECK$ [=] "Y" [OR] CHECK$ [=] "N"
                                                └<>から=へ変更
```

やや混乱しましたか？

まとめると、次のようになります。

■**WHILEの時**

・満たさなくなるまで繰り返す

・変数CHECK$が“Y”以外（<>）であり、かつ（AND）“N”（<>）以外のときはループ終了

■**UNTILの時**

・満たすまで繰り返す

・変数CHECK$が“Y”である（=）、もしくは（OR）“N”である（=）場合にはループ終了

4　強制終了（EXIT DO）

DO...LOOP文は、これまで解説しただけでも柔軟性のある仕様となっていますが、“EXIT DO”を使えば、さらに柔軟性が増します。

EXIT DOは

強制的にDO...LOOP文を終了

させることができるステートメントです。

次のサンプルプログラムは、プログラム内部で設定した変数の内容と、ユーザーが入力したキーを当てるのに、どれだけかかるか計測するプログラムです。

もしプログラム内で設定した数値とユーザーが入力した数値が等しくなれば、EXIT DOを使ってループを終了させることができます。

●ソースリストPRG-NO34.BAS

```
CLS
PRINT "Input data [    ]  TIME :"
ORG$ = "c"

DO
    LOCATE 1, 26

    PRINT COUNT
    COUNT = COUNT + 1

    K$ = INKEY$

    IF K$ <> "" THEN ←── 変数K$の内容が"" 以外のときはここに入る
        COLOR 6
        LOCATE 1, 14
        PRINT K$ ←── 入力したキーの表示
        BEEP
        COLOR 7
        IF ORG$ = K$ THEN ←── 変数ORG$の内容とK$の内容を比較
            PRINT "Congratulations !" ←── 等しければ
            EXIT DO                         次のステートメント実行
        END IF
    END IF
LOOP UNTIL COUNT > 2000 ←── 変数COUNTの内容が2000を越えると終了
```

変数ORG$が、内部に設定した値です。

これをユーザーが当てようというのですが、作成した本人は変数ORG$が何であるか知っているので、あまりおもしろくありません。こんな時は、“乱数”という手法を用いるとコンピュータがめちゃくちゃな数値を吐き出してくれるので、作成した本人にも分からないおもしろい数値を設定できると思います。

乱数については、Quick BASICのリファレンスマニュアルに素晴らしいサンプルがあるので、参考にしてください。

実行してみましょう。

◉インタプリタによる実行結果1

```
Input data [ d ]  TIME : 2000
```

◉インタプリタによる実行結果2

```
Input data [ c ]  TIME : 514
Congratulations !
```

第10章

サブルーチン

ここでは、サブルーチンを学びます。

BASICに限らず、サブルーチンの概念は、コンピュータ言語においては変数などと同じくらい重要な概念となります。

どんな小さなことでも、おろそかにできないのがコンピュータ言語ですが、その全てをいちいちすべてソースの中に記述していたら大変な労力です。

そこで、サブルーチンという概念が取り入れられています。

同じことをする仕事は、あらかじめそのプログラムを作っておいて、その場その場で、それを呼び出して使うというのが、サブルーチンの方法です。

ここで、サブルーチンがなんであるか、また、実際にはどのように使うのかを学び取ってください。

10・1

サブルーチンの概念

同じ処理を分担する小さなプログラム

1 サブルーチンとは

サブルーチンとは、ある仕事を分担した小さなプログラムのことをさします。

大きなプログラムでは、例えば「Y/Nで答えてください：」というメッセージを出力させたり、この時点で入力できるキーを“Y”と“N”だけに制限するなどのように、何度何度も登場する同じ処理が必要になります。

このような処理を、サブルーチンという概念なしで行なうと、いちいちメインのプログラムの中に、同じ処理を行なうステートメントを書き込まなくてはなりません。

これでは効率的ではありませんし、また、プログラム自体も長くなってしまいます。

そこで、コンピュータ言語の世界では、サブルーチンという概念を取り入れ、同じ処理はいつもそのサブルーチンに分担をさせます。

その処理が必要とされた時には、そのサブルーチンを呼び出して使うのです。

2　サブルーチンの種類

Quick BASICでは、いくつかのサブルーチンが用意されており、状況に応じて使い分ける必要があります。

①GOSUB

従来のBASICで使われていた、サブルーチンの呼び出しです。

この方法は、「行番号」もしくは「ラベル」という名前でサブルーチンを呼び出します。

ここに挙げる３つのサブルーチンの中で、もっとも簡単なサブルーチンの呼び出し方法です。

しかし、この方法では、サブルーチンへの変数の引き渡しができないため、あまり勧められませんが「サブルーチンがなんであるか」を理解するには、使い方が簡単なぶん向いているといえるでしょう。

②SUB

「SUBプロシージャ」と呼びます。

これは、GOSUB文とは若干違い、変数を引き渡して計算を行なったりすることができます。

③FUNCTION

「FUNCTIONプロシージャ」と呼びます。

これは、仕事をまとめて登録しておくという意味ではサブルーチンとして考えられますが、Quick BASICでは、関数を作成する場合に、このFUNCTIONプロシージャを主に使います。

この「FUNCTIONプロシージャ」が、Quick-BASICの最大の特長ともいって過言ではないと思います。

従来のBASICでは、ステートメントや関数を自分で拡張することは不可能でした。しかし、FUNCTIONプロシージャを使えば簡単に言語自身の機能をアップさせることができます。

では、それぞれの使い方を見ていきましょう。

10・2
GOSUB

使い方の制約が少ない伝統的なサブルーチン

1 GOSUBの記述方法

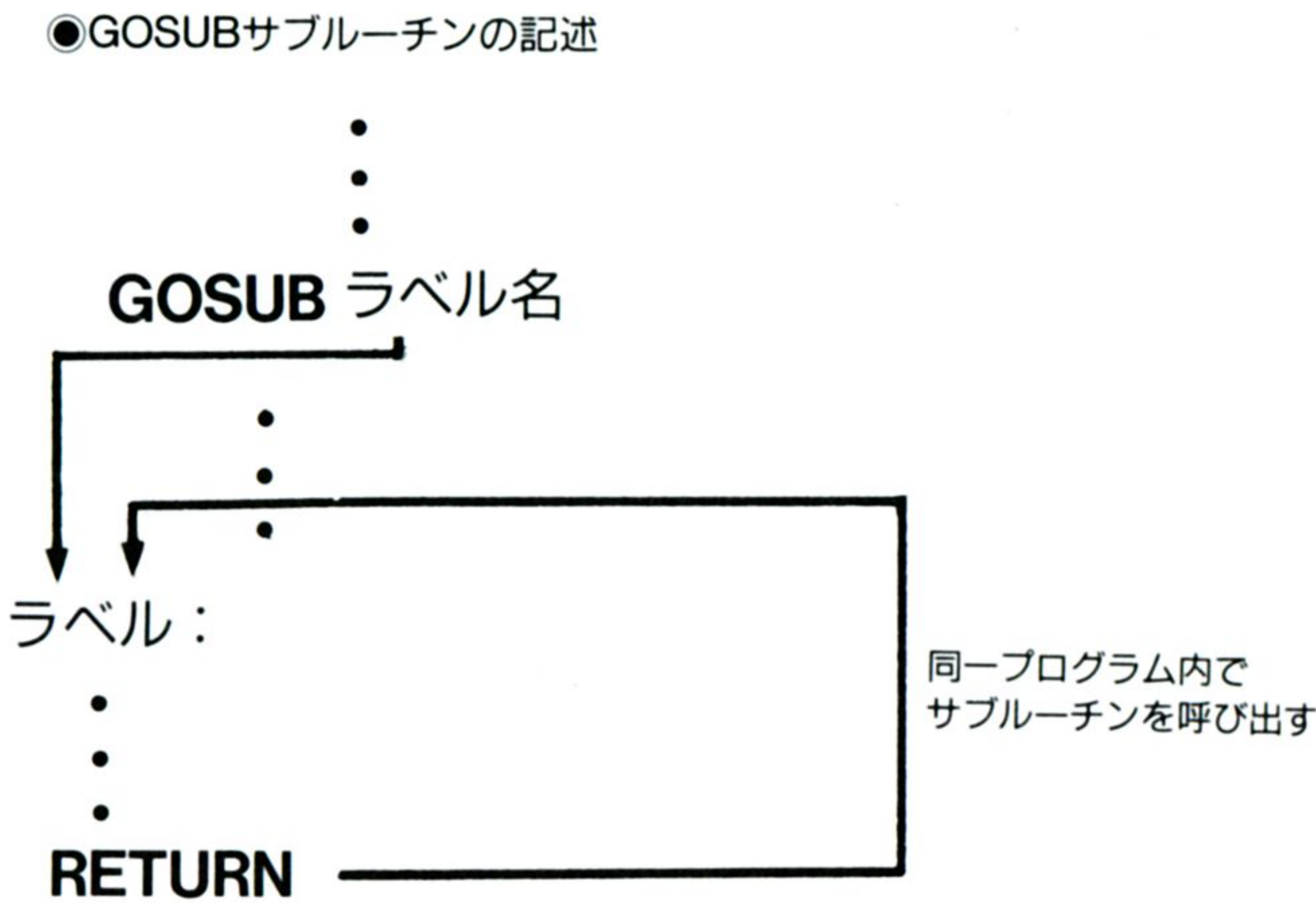
3つ紹介したサブルーチンの中で、もっとも伝統的であり、かつ使い方の制約が少ないサブルーチンです。

ただ、使い方が簡単なぶん機能が少なく、また変数の扱いに若干問題があります。

したがって、処理をある程度理解したらGOSUB文を使わないで、SUBプロシージャやFUNCTIONプロシージャを使った方がいいでしょう。

●GOSUBサブルーチンの記述

```
        ・
        ・
        ・
GOSUB ラベル名
      ・
      ・
      ・
ラベル：
  ・
  ・
  ・
RETURN
```

同一プログラム内で
サブルーチンを呼び出す

■ラベルを使ってサブルーチンを呼び出す

さて、GOSUB文は、行番号で分岐先を指定できる機能を持っています。

本書では行番号は扱わないので、ラベルを使ったサブルーチンの呼び出しを解説します。

■ラベルに使える文字

ラベルは、BASICのステートメントや関数などで使われていない文字ならば何でも使用可能です。

ただし、ラベル名の先頭に文字がくる必要があり、40字以内（半角で）の文字と決っています。これは、変数の扱いと同じです。

■「ラベル：」からRETURNステートメントまでを実行

プログラムを実行中にGOSUBのステートメントを見つけると、BASICは無条件に“ラベル：”に分岐します。そして、“ラベル：”からRETURNステートメントまでを実行して、呼び出したGUSUB文まで戻ってきます。

■サブルーチンは何度でも使える

呼び出すサブルーチンは、何度でも使うことができます。また、サブルーチンの中から、別のサブルーチンを呼び出すことができます。

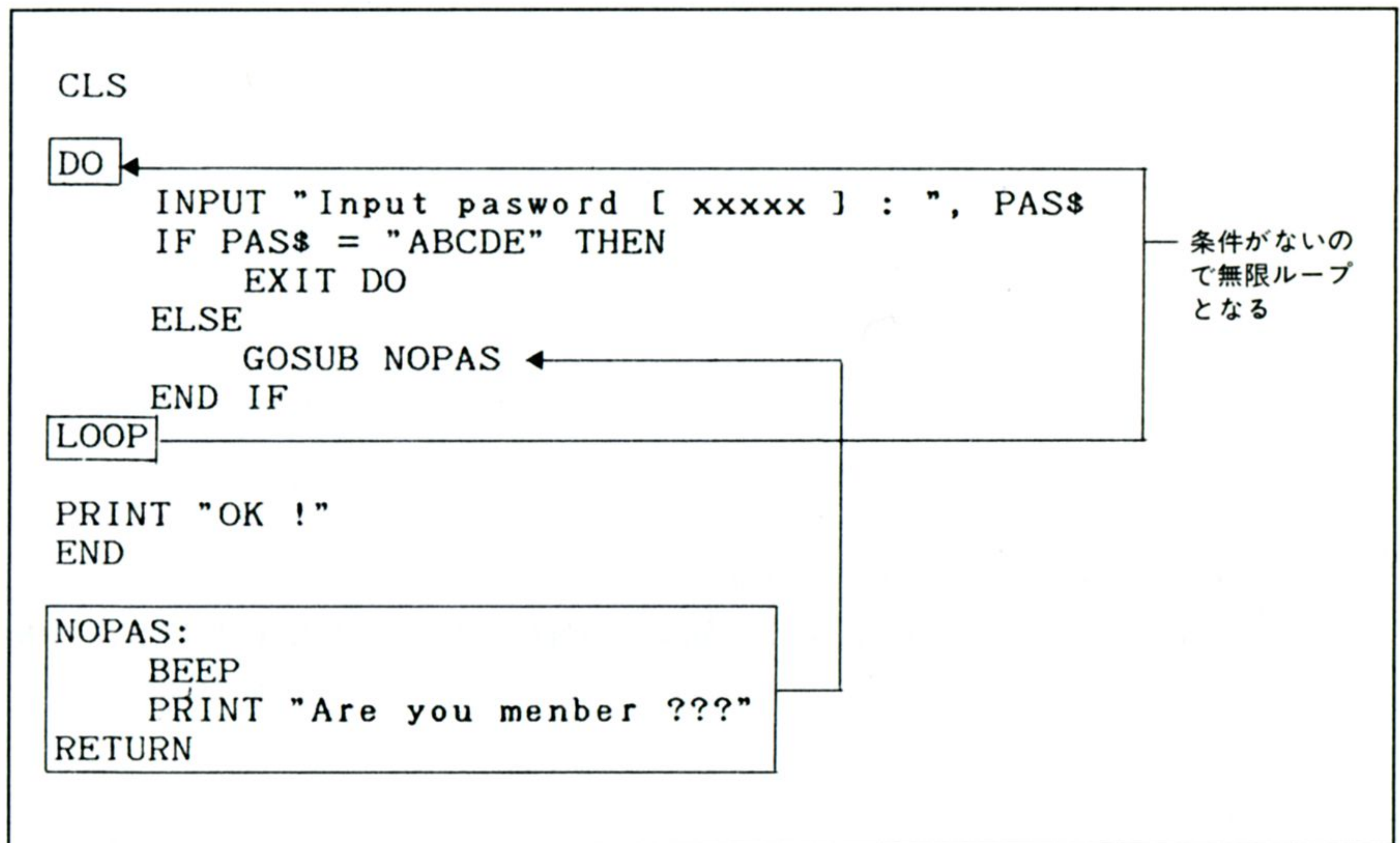
ただし「別のサブルーチンを呼び出す」と、自分で作成したプログラムであるにも関わらず、複雑なプログラムとなってしまうので、なるべく別の方法を考えてください。

GOSUB文は、無条件分岐をするため、どんな行にも(若干の制約はあります）飛ぶことができるため、呼び出したところに戻らないで終了してしまうこともあります。また、実行するつもりのなかった行を実行することもあります。

2 サブルーチンGOSUBの使い方

さぁ、実際にプログラミングを行ないながらGOSUBを収得していきましょう。

◉ソースリストPRG-NO35.BAS

```
CLS

DO
    INPUT "Input pasword [ xxxxx ] : ", PAS$
    IF PAS$ = "ABCDE" THEN
        EXIT DO
    ELSE
        GOSUB NOPAS
    END IF
LOOP

PRINT "OK !"
END

NOPAS:
    BEEP
    PRINT "Are you menber ???"
RETURN
```


このプログラムは、5桁のパスワードを入力させ、入力したパスワードが正しいかを判断させています。パスワードが、あらかじめ設定したパスワードと違う場合には、GOSUB文でラベルNOPASに飛び、ビープ音と共にエラーメッセージを表示し再入力をさせます。

今までのプログラムの場合よりも、スッキリしたでしょう。

通常のプログラミングでは、このように、仕事ごとにプログラムを分けて記述します。このような記述方法を構造化ともいいます。

また、ENDステートメントを入れて終りを明確にしてください。

これを忘れると、メインプログラムが終了したのち、サブルーチンに入っていってしまいますので注意が必要です。

実行してみましょう。

●インタプリタによる実行結果

```
Input pasword [ xxxxx ] : dhiFO
Are you menber ???
Input pasword [ xxxxx ] : 145dW
Are you menber ???
Input pasword [ xxxxx ] : ABCDE
OK !
```

もう1つ、GOSUBサブルーチンを使ったプログラムを作ってみましょう。

このプログラムは、入力した数値で文字の色を設定させるプログラムです。

●ソースリストPRG-NO36.BAS

```
CLS

DO
    INPUT "Input number [ 0-END, 1-7 ] : ", COL
    IF COL >= 1 AND COL <= 7 THEN
        GOSUB CSET
    END IF
LOOP UNTIL COL = 0
PRINT "End..."
END

CSET:
    COLOR COL
    PRINT "Setting color..."
    COLOR 7
RETURN
```

"0"を入力するとループを抜ける

入力を行なうとサブルーチンCSETでカラーの設定を行ないます。

カラー設定の処理だけサブルーチン化しているので、メインルーチンがスッキリしたと思います。

入力した数値は変数COLに格納され、その値を元に、サブルーチンCSETでカラーの設定を行なっています。ここで注意していただきたいのが「変数の値がサブルーチンに影響する」ということです。

実行してみましょう。

●インタプリタによる実行結果

```
Input number [ 0-END, 1-7 ] : 1
Setting color...   ←――――――――― 青で表示
Input number [ 0-END, 1-7 ] : 2
Setting color...   ←――――――――― 緑で表示
Input number [ 0-END, 1-7 ] : 3
Setting color...   ←――――――――― 水色で表示
Input number [ 0-END, 1-7 ] : 9 ←―― 入力オーバーなので再入力
Input number [ 0-END, 1-7 ] : 0 ←―― 終了
End...
```

GOSUBでの変数の扱いは、あくまでもメインルーチンと同等です。つまり、メインルーチンで使用した変数は、サブルーチンでも同じように使用することになります。

このことは、この次に解説するSUBプロシージャやFUNCTIONプロシージャと密接な関わりがありますので、一応覚えておいてください。

10・3
SUBプロシージャ
変数の引き渡しができるサブルーチン

1　SUBプロシージャの記述方法

SUBプロシージャの概念は、ほとんどGOSUBと同じように無条件分岐しますが、変数の引き渡しが出来る点が違います。
SUBプロシージャは、次のように記述します。

●SUBプロシージャの記述

〈メインルーチン〉

```
DECLARE SUB SUBプロシージャ名(変数名, …)
  :
CALL SUBプロシージャ名(変数名, …)
  :
```

```
SUB SUBプロシージャ(変数名, …)
  :
END SUB
  :
```

■変数の引渡し

変数の引渡しとは、変数をSUBプロシージャに引き渡して、演算などを行なわせることを意味します。

つまり、メインルーチンで使っていた変数を、SUBプロシージャに引き渡して演算をさせた後、その値をメインルーチンで受け取れるわけです。一見GOSUBと似ていますが、変数の扱い方が違います。

■メインルーチンと無関係に変数を扱える

GOSUBの場合には、GOSUB...RETURN内で初めて使う変数であろうと、メインルーチンの変数であろうと同じように使用できますが、SUBプロシージャの場合には、SUB...END SUB内で初めて使用する変数はメインルーチンと全く無関係です。

このことをローカル変数と呼びます（ローカル変数の反対の意味を持つのがグローバル変数）。ローカル変数とは、変数の内容を保持する領域に制限があることを意味します。つまり、SUBプロシージャの場合には、「呼び出した時にだけ有効な変数」となります。

■サブルーチンを、全く別の空間に持つ

もう1つGOSUBと決定的に違うこととして、サブルーチンを同じファイル内に持つのではなく、「全く別の空間に持つ」という概念を持っています。

実際は同じファイル内にSUBプロシージャを持っていますが、プログラムを作成する段階で“SUB...”と入力すると、今まで作成していたエディタとは別にウィンドウが開き、全く別の空間としてSUBプロシージャを作成します。

■DECLAREステートメントで使用宣言をする

SUBプロシージャは、使用宣言を行なう必要があります。

GOSUBのようにいきなりサブルーチンを使うのではなく、SUBプロシージャを使う前にDECLAREステートメントで、使用するSUBプロシージャの使用宣言を行ないます。

この宣言を行なうと、CALLステートメントでSUBプロシージャを呼び出すことなく、単に呼び出すSUBプロシージャ名を書き込むだけで呼び出すことができるようになります。

2　SUBプロシージャの使い方

次のプログラムは、もっとも基本的なSUBプロシージャを使った例です。

変数の引き渡し

◉ソースリストPRG-NO37.BAS

```
DECLARE SUB CALC (A!,, B!)   ← SUBプロシージャCALCの使用宣言
CLS

A = 1
B = 2
PRINT "MAIN : A ="; A; " B ="; B
CALL CALC(A, B)
PRINT

A = 3
B = 4
PRINT "MAIN : A ="; A; " B ="; B
CALL CALC(A, B)
PRINT

A = 5
B = 6
PRINT "MAIN : A ="; A; " B ="; B
CALL CALC(A, B)
                             ← ENDステートメントがなくても正常に終了
```

ブロックごとに変数の内容を変更
SUBプロシージャCALC呼び出し

◉SUBプロシージャCALC

```
SUB CALC (A, B)
    A = A * 10
    B = B * 100
    PRINT "SUB  : A ="; A; "B ="; B
END SUB
```

ここでは、変数の引き渡しの動作の確認を行なっています。

SUBプロシージャCALCでは、それぞれの渡された変数に10もしくは100をかけて、それをそのまま呼び出された側に返しています。

SUBプロシージャは、変数を受け取る際の変数名を、自由に命名することができます。ここでは、混乱を避けるために同じ変数名で使用しました。

●SUBプロシージャの変数を変更した例

```
SUB CALC (TEMP1, TEMP2)
    TEMP1 = TEMP1 * 10
    TEMP2 = TEMP2 * 100
    PRINT "SUB  : A ="; TEMP1; "B ="; TEMP2
END SUB
```

ただし、変数名を変更した場合には、メインルーチン側の先頭で宣言しているDECLARE文の変数も変更しなくてはなりません。

実行してみましょう。

●インタプリタによる実行結果

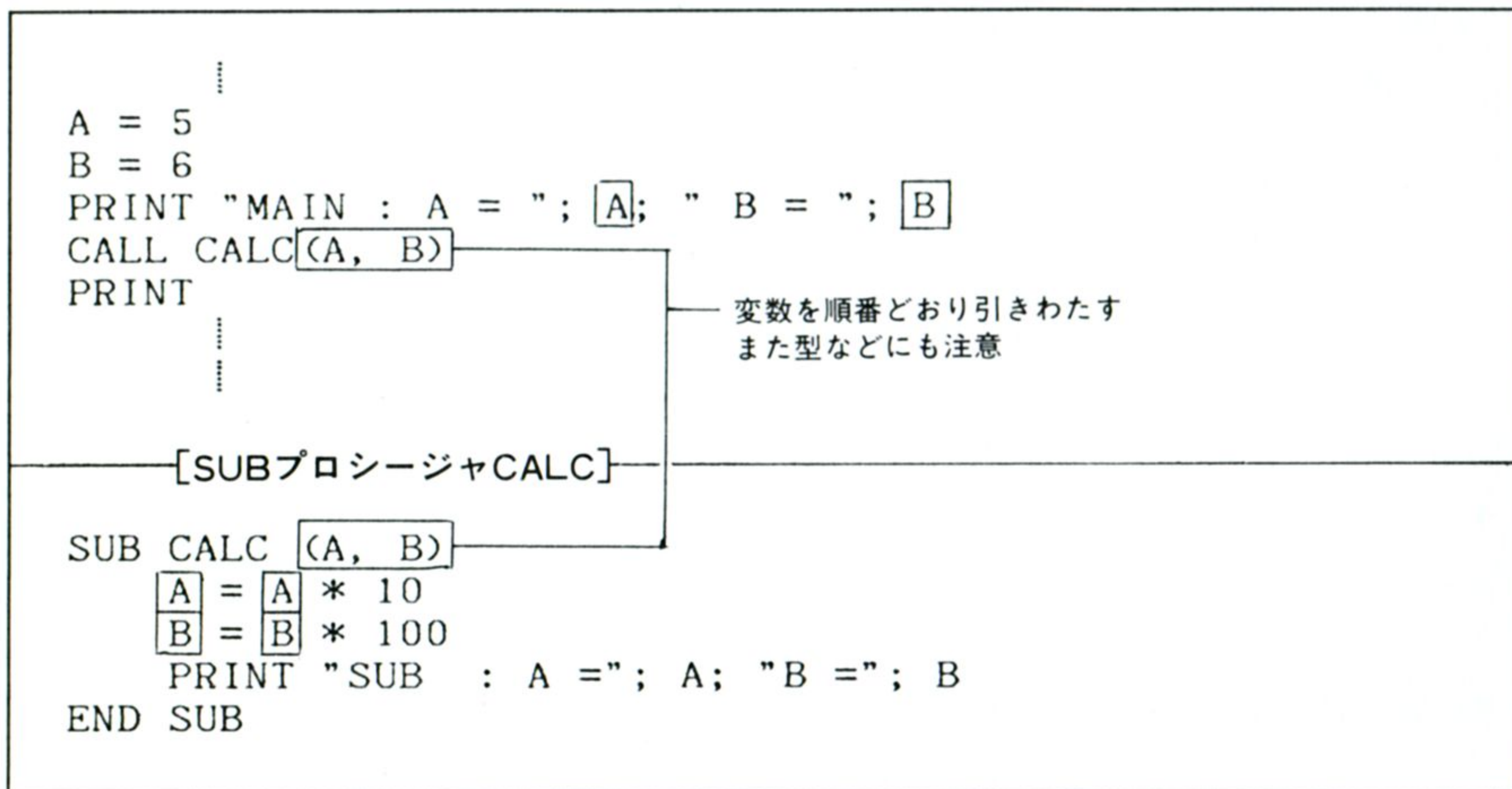
```
MAIN : A = 1  B = 2
SUB  : A = 10 B = 200

MAIN : A = 3  B = 4
SUB  : A = 30 B = 400

MAIN : A = 5  B = 6
SUB  : A = 50 B = 600
```

次に、SUBプロシージャを呼んだときの変数の流れなどを見ていくことにしましょう。

SUBプロシージャに変数を渡すには、まず、メインルーチン側から渡す変数の数、型（文字型、数値型など）をSUBプロシージャと合わせる必要があります。

◉変数の関係

```
        ⋮
A = 5
B = 6
PRINT "MAIN : A = "; [A]; " B = "; [B]
CALL CALC[(A, B)]
PRINT
        ⋮
        ⋮
──────[SUBプロシージャCALC]──────
SUB CALC [(A, B)]
    [A] = [A] * 10
    [B] = [B] * 100
    PRINT "SUB  : A ="; A; "B ="; B
END SUB
```

変数を順番どおり引きわたす
また型などにも注意

“()”の中に書く変数の情報を合わせて使ってください。

これは実に重要なことで、SUBプロシージャを使う場合の要となります。

変数を引き渡すとして、SUBプロシージャを宣言すると、以降変数を引き渡したくなくても、必ず変数を引渡し形でしか使えないので注意が必要です。

SUBプロシージャでは、DECLARE文で使用宣言を行なう必要があると述べましたが、実は、プログラム中で“SUB ...”と入力すると自動的にQuick BASICがDECLARE文を作成してくれます。

SUBプロシージャを使えば、インタプリタで実行したり、ディスクに保存したりすることで、自動的にSUBプロシージャの宣言を行なってくれます。

ソースリストをよく見ると、引き渡す変数名を明示していることに注意してください。

DECLARE SUB　(X!, Y!)

変数名には“!”が付けられています。これは、変数がもつ数値型とか文字型のほかに、型宣言文字と呼ばれ、その変数の扱える数値の大きさを明確にしているのです。

“!”が付く変数名は、「小数点以下が15桁以内の数値」であることを意味しています。

3　ローカル変数

今度は変数の持つ有効範囲である、ローカル変数を理解しましょう。

●ソースリストPRG-NO38.BAS

```
DECLARE SUB SAMPLE ()
CLS
A = 2 ┐
B = 3 ┘── 変数A，Bにそれぞれ数値を代入
PRINT "MAIN  A ="; A; "   B ="; B ←── SUBプロシージャを呼び出す前に表示
CALL SAMPLE
PRINT "MAIN  A ="; A; "   B ="; B ←── SUBプロシージャを呼び出した後に表示
```

●SUBプロシージャ・SAMPLE

```
SUB SAMPLE
    A = 200 ┐
    B = 300 ┘── SUBプロシージャの中で同じ名前の変数に数値
    PRINT "SUB  A ="; A; " B ="; B
END SUB
```

◉インタプリタによる実行結果

```
MAIN   A = 2     B = 3   ────── SUBプロシージャを呼び出す前の変数の値
SUB    A = 200   B = 300 ───── SUBプロシージャ内で変数に代入をした値
MAIN   A = 2     B = 3   ───── 呼び出したときの値と同じ点に注意
```

まず、SUBプロシージャを呼び出す前に変数A、Bに値を代入し、それを表示させます。たしかに、変数Aには2、変数Bには3が入っていますね。

そして次に、SUBプロシージャSAMPLEでメインルーチン側で使用している変数と同じ名前の変数にそれぞれ数値を代入します。

ここでは、変数Aに200、変数Bに300を代入しました。実行結果でも、たしかに代入した値が入っています。

さて、問題はこの次です。

変数のローカル化

通常のサブルーチンでこのようなことを行なうと、呼び出した側の変数の値も変化します。同じ名前の変数に値を代入すれば、当然呼び出した側の値も変化するはずですが、実行結果を見ると、呼び出した側の変数の値はSUBプロシージャSAMPLEを呼び出す以前の値と同じです。これが、ローカル変数なのです。

今まで、変数を扱うときサブルーチン側で使う変数とメインルーチン側で使う変数をたえず気にしながらプログラミングを行ないましたが、Quick BASICでは、変数のローカル化が行なえるので、同じ名前の変数でも気にせずそれぞれのルーチンで使えます。

このことは、実に重要なことです。

例えば、カウントを取るようは変数は、たいがい“COUNT”などと命名しますが、このような変数は、ちょっとしたプログラムならば頻繁に使用します。今までのように、サブルーチンで使っている変数もメインルーチンに影響するようなときにはどうなるでしょうか。

そうです。変数がかち合わないように、別々の変数として命名を工夫しなくてはなりません。

このようにすると、他のプログラムでも使えるようなサブルーチンを作成しても、「どんな変数を使っているのか？」と、気を使う必要が出てきますね。

しかし、SUBプロシージャのように、“変数の有効範囲”がそのSUBプロシージャの中だけならば、自由に変数を命名することができます。

これがローカル変数の概念なのです。

また、ローカル変数は、SUBプロシージャだけでなく次に解説するFUNCTIONプロシージャでも同じように、変数の有効範囲を特定させることができます。

10・4

FUNCTIONプロシージャ

関数と同じように「値を返す」サブルーチン

1　FUNCTIONプロシージャの記述方法

FUNCTIONプロシージャは、SUBプロシージャと機能、使い方など、ほとんど同じですが、式などの中に混合して使えます。

つまり、FUNCTIONプロシージャは、既存の関数と同じように使えるわけです。また、Quick BASIC付属のクイックライブラリを使えば、いちいち関数宣言を行なわなくても、全く関数と同じように使えます。

FUNCTIONプロシージャは、次のように記述します。

●FUNCTIONプロシージャの記述

〈メインルーチン〉

```
DECLARE FUNCTION FUNCTIONプロシージャ名(変数名, …)
   :
CALL FUNCTIONプロシージャ名(変数名, …)
   :
```

〈FUNCTIONプロシージャ〉

```
FUNCTION FUNCTIONプロシージャ名(変数名, …)
   :
END FUNCTION
   :
```

2　FUNCTIONプロシージャの使い方

では、実際にプログラミングを行なって、FUNCTIONプロシージャを理解していきましょう。

●ソースリストPRG-NO39.BAS

```
DECLARE FUNCTION MENSEKI! (TEIHEN!, TAKASA!)
CLS
INPUT "TEIHEN  : ", A
INPUT "TAKASA  : ", B
PRINT "MENSEKI :"; MENSEKI(A, B)
```

計算結果をここに返す
FUNCTIONプロシージャ側の変数名に注意

●FUNCTIONプロシージャMENSEKI

```
FUNCTION MENSEKI (TEIHEN, TAKASA)
    MENSEKI = TEIHEN * TAKASA / 2
END FUNCTION
```

それぞれに対応

原則的に、SUBプロシージャと同じです。

しかし、SUBプロシージャとの大きな違いは、「値を返す」という点です。

SUBプロシージャでは、Quick BASICのステートメント中で使用するとエラーとなりますが、FUNCTIONプロシージャでは、ステートメント中で使用してもなんら問題がありません。

また、SUBプロシージャで使用していたCALLステートメントを使用していません。実は、FUNCTIONプロシージャもCALLステートメントを使うことができますが、SUBプロシージャと分けて考えてもらうために、あえてCALLステートメントを使用しないでFUNCTIONプロシージャを解説しました。

さて、プログラム自身は簡単なことしか行なわせていないので、簡単に理解できると思います。

FUNCTIONプロシージャで行なった計算結果は、「FUNCTIONプロシージャとして付けた名前に代入する」ことで、メインルーチンに返すことが出来ます。

FUNCTIONプロシージャMENSEKIの中で演算を行ない、その結果を、MENSEKIという変数に代入してやればいいのです。

実行してみましょう。

●インタプリタによる実行結果

```
TEIHEN  : 10
TAKASA  : 15
MENSEKI : 75
```

理解度を増すために、もう１つプログラムを作ってみましょう。

●ソースリストPRG-NO40.BAS

```
DECLARE FUNCTION MONTH$ (TEMP!)
CLS
DO
    INPUT "Month : ", M
LOOP WHILE "NULL" = MONTH$(M)
PRINT MONTH$(M)

END
```

$マークに注目

●FUNCTIONプロシージャMONTH$

```
FUNCTION MONTH$ (TEMP) '
    SELECT CASE TEMP

        CASE 1
            MONTH$ = "January"
        CASE 2
            MONTH$ = "February"
        CASE 3
            MONTH$ = "March"
        CASE 4
            MONTH$ = "April"
        CASE 5
            MONTH$ = "May"
        CASE 6
            MONTH$ = "June"
        CASE 7
            MONTH$ = "July"
        CASE 8
            MONTH$ = "August"
        CASE 9
            MONTH$ = "September"
        CASE 10
            MONTH$ = "October"
        CASE 11
            MONTH$ = "November"
        CASE 12
            MONTH$ = "December"

        CASE ELSE
            MONTH$ = "NULL"
    END SELECT
END FUNCTION
```


プログラム中の注意書きに注意してください。

先ほどのプログラムは、返される値が数値だったのに対し、今度は“文字が返される”点が大きく違います。

プログラムを見ても分かる通り、文字で返される値の場合には、“$”マークを呼び出すサブプログラム名に付けなくてはなりません。

また、ちょっとしたテクニックとして、入力された数値をFUNCTIONプロシージャの中で判断させました。

月名など、決った数値以外の入力が入った場合のエラーチェックを含ませることにより、より汎用的なFUNCTIONプロシージャができるでしょう。

実行してみましょう。

●インタプリタによる実行結果

```
Month : 15
Month : 11
November
```

第 11 章

高度な変数の利用

私たちは、様々な形で変数を使ってきました。今まで使ってきた変数の区別は、大別すると数値型と文字型だけでしたが、Quick BASICでは、

定数（決った数）を扱う変数

配列、ローカル変数をグローバル変数に変更

など、様々な変数の扱い方があります。

本章では、これら変数の違いを学びます。

11・1
定数

特定の値に名前を付けて使う

1　定数とは

定数とは、「決った数」のことをいいます。

1時間は60分、1年は12カ月などのように、世の中には決った数を名前で呼ぶことがあります。

これが定数の概念です。

Quick BASICでもこの概念で定数を扱うことができます。つまり、決った数に名前を付けて使用することができるのです。

2　定数の宣言

定数は、次のようにして宣言します。

●●定数の宣言●●

```
┌───定数の宣言
CONST 変数名 = 数値もしくは文字列
      └─定数の名前  └─定数の内容
```

・CONSTステートメント

従来の変数を扱うように行ないますが、定数にするには、CONSTステートメントを先頭に付けます。

・変数名（定数として扱うときの変数名）

定数は、数値型などの変数と同じように使います。したがって、定数の使用宣言を行なうときに、定数の変数名を決めてやります。

また、定数にも扱える変数が決っており、数値型ならば何も付けずに宣言し、文字型ならば＄マークを付けて宣言します。

・定数の内容

扱う定数の内容そのものです。

基本的には通常の変数のように扱われますが、定数として宣言された変数の場合には、代入などができません。つまり、変数の内容の変更が許されないのです。また、定数はSUBプロシージャやFUNCTIONプロシージャなどのモジュール内でも使用できます。

ただし、もし、定数にした変数の内容を変更しようとすると次のようなエラーが出ますので注意してください。

宣言の重複は許されません（ERR=10)

一見、変数と似ており、定数にする必要性がないように思われますが、次のようなメリットがあります。

①値の変更ができない

値の変更ができないので、不注意でバグを作ることがなくなる。

②通常の変数よりも効率的なコードを生成

無駄なメモリを食わずにすむ。

③プログラムの保守が簡単

定数にした変数の値を先頭で変更するだけで、範囲などが自由に設定できる。

これだけでは何をいっているのか分からないと思いますので、実際のプログラムを通して定数を理解することにします。

3 定数の使い方

次のようなプログラムをつくりましょう。

◉ソースリストPRG-NO41.BAS

```
CLS
CONST HI = 10
CONST LOW = 1

DO
    INPUT "Please input LEVEL : ", LEVEL
LOOP WHILE LEVEL < LOW OR LEVEL > HI

PRINT "Your LEVEL is "; LEVEL
```

最大値をHIという定数にし、最小値をLOWという定数にしています。最大値と最小値の間だけ数値入力を行なわせ、それ以外は全て再入力になります。

このプログラムの利点は、"CONST ..."となっている部分の数値を変更するだけで、最大値と最小値の値を変えることができる点です。

実行してみましょう。

◉インタプリタによる実行結果

```
Please input LEVEL : 11
Please input LEVEL : 0
Please input LEVEL : 4
Your LEVEL is 4
```

こういったことは、一般でもよく使われることです。例えば、「○○点以下は警告、××点以下は赤点」などのように、警告の点数のラインが変動するような場合には、"CONST KEIKOKU = 40" などのようにすればいいのです。

次に、このような例のプログラムを書いてみましょう。

◉ソースリストPRG-NO42.BAS

```
CLS
CONST KEIKOKU = 45 ――――― 警告ライン
CONST AKA = 30 ――――― 最低ライン
CONST NINZU = 5 ――――― 入力を行う人数
X = 5 ――――― 棒グラフの表示位置の初期値
LOCATE 3, 1
PRINT "NAME                                 20        40        60        80       100"
PRINT "--------------+---------+---------+---------+---------+---------+"
  ――― 棒グラフの表題
DO
    LOCATE 1, 1
    INPUT "Name  : ", NAMAE$ ――― データ入力
    INPUT "Score : ", TEN
    IF TEN < KEIKOKU AND TEN > AKA THEN
        COLOR 6
    END IF ――― 入力された値の判断
    IF TEN < AKA THEN
        COLOR 4
    END IF
    LOCATE X, 1
    PRINT NAMAE$, ":"; STRING$(INT(TEN / 2), "*") ――― データの表示と点数の棒グラフ化
    LOCATE 1, 8
    PRINT "                              " ――― データの入力領域を空白で埋める
    LOCATE 2, 8
    PRINT "                              "
    COUNT = COUNT + 1 ←―― 人数のカウント
    X = X + 1 ←―― 棒グラフの表示領域を1行下にずらす
    COLOR 7 ←―― 変更された色を元に戻す
LOOP UNTIL COUNT = NINZU
  └―― 変数COUNTが定数NINZUと等しくなるとループを抜ける
```

ここでの定数のメリットは、警告となる点数のラインと赤点となる点数のライン、それと人数を変更できる点にあります。

例えば、ここでは、“CONST NINZU = 5” としてあるので入力できる人数は5人で終りますが、これを “CONST NINZU = 10” などとすれば、簡単に扱える人数を増やすことができます。

●インタプリタによる実行結果

```
Name  :
Score :
NAME                   20        40        60        80       100
--------------+---------+---------+---------+---------+---------+
SAWADA        ;**************************************************
KIMURA        ;***
WATANABE      ;**********************
YOKOBE        ;*******************************
MARUYAMA      ;*
```

11・2
配列
数の分からない要素を使う

1　配列とは

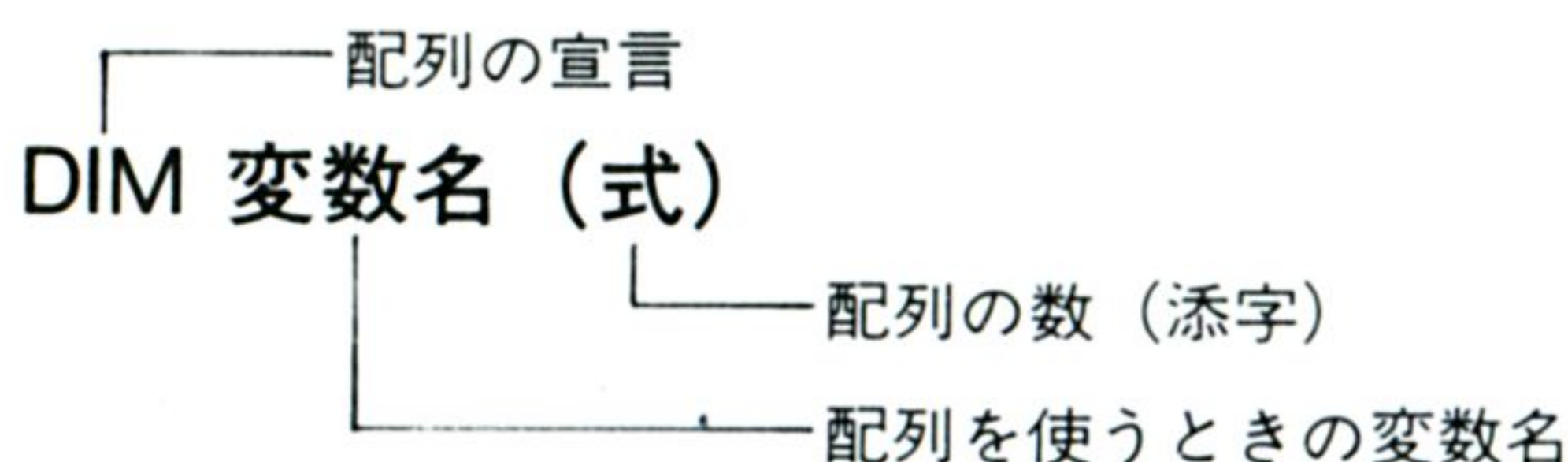
配列とは「決った規則に従う変数の集まり」として考えることができます。

数が分かっていない未知の要素を使うときなど、配列が役に立ちます。

2　配列の宣言

配列の宣言方法は、次のように行ないます。

●●配列の宣言●●

```
┌──────配列の宣言
DIM 変数名（式）
      │     └────配列の数（添字）
      └──────────配列を使うときの変数名
```

・DIM（配列の宣言）

配列は、実際に使う前に宣言をしなくてはなりません。宣言は、ここで示しているDIMステートメントで行なわれます。

・変数名（配列を使うときの変数名）

配列は、数値型などの変数と同じように使います。したがって、配列の使用宣言を行なうときに、配列の変数名を決めてやります。

また、配列にも扱える変数が決っており、数値型ならば何も付けずに宣言し、文字型ならば＄マークを付けて宣言します。

・式（配列の数［式は添字ともいう］）

配列に使う、数量を指定します。通常、ここには数値もしくは変数が入りますが、応用的な使い方として、数量を範囲で指定することも可能です。

3　配列の使い方

さっそく、配列を使ったプログラミングをしてみましょう。

◉ソースリストPRG-NO43.BAS

```
CLS
DIM A(4)←――――配列の宣言。ここでは5個の配列が使える

A(0) = 1 ┐
A(1) = 2 │
A(2) = 3 │それぞれの配列に数値を代入
A(3) = 4 │
A(4) = 5 ┘

PRINT A(0); A(1); A(2); A(3); A(4)←――――配列の内容を表示
```

配列の添字は、通常は“0”から始まるので、式となるところに“4”を指定すると“0”～“4”までということになるので、扱える範囲は“5”になります。

配列に代入を行なう場合には、通常の変数と同じく、“A（1）＝2”のように行ないますが、違うのは、変数名の後についた（式）です。つまり、配列A（1）と配列A（2）は別の変数ともいえます。

配列を表示させるには、変数名についている「(式)」をステートメント中に書きこめば良いわけです。

実行してみましょう。

●インタプリタによる実行結果

```
 1       2       3       4       5
```

次に、文字型の変数を扱う例を行なってみることにします。

●ソースリストPRG-NO44.BAS

```
CLS
DIM MEN$(3)                                ┌──注意
FOR I = 0 TO 3                             ▼
    INPUT "Please name : ", MEN$(I)
NEXT I
PRINT MEN$(0), MEN$(1), MEN$(2), MEN$(3)
```

先ほど扱った配列と違っているのは、配列の変数名に＄マークが付くことです。この点を満たせば、文字列も配列として扱えます。

これだけの名前を登録するには、これまでの方法ですと、5つの変数が必要でしたが、変数を添字に使えば、1つの配列で個別のデータを全く別のところに確保できます。

実行してみましょう。

●インタプリタによる実行結果

```
Please name : SUZUKI
Please name : SAITOU
Please name : WADA
Please name : MATUDA
SUZUKI        SAITOU        WADA          MATUDA
```

ただし、添字に変数を使うときには、かならず扱う数量を考慮に入れてください。

例えば、次のような例はエラーとなります。

```
DIM MEN$ (X)
```

変数Xには、何も入っていない状態なので“0”となってしまいます。このような使い方をするときは、添字の変数にあらかじめ数値を代入することでエラーを回避できます。

```
X = 10
DIM MEN$ (X)
```

次は、配列の範囲を指定する方法を試みてみましょう。

●ソースリストPRG-NO45.BAS

```
CLS
DIM A(-3 TO 3)

FOR I = -3 TO 3
    A(I) = I + 4 ←――①配列A(I)の最初の添字は“-3”
    PRINT A(I)
NEXT I
```

添字の範囲を指定する場合には“TO”を使います。

-3～+3までの範囲を添字にするには、“DIM A (-3 TO 3)”とします。

●インタプリタによる実行結果

```
1
2
3
4   配列A(I)の扱える最大の数は7ということが分かった
5
6
7
```

11・3

変数のグローバル化

全ての領域に対して効力を持つ変数

1　グローバル変数

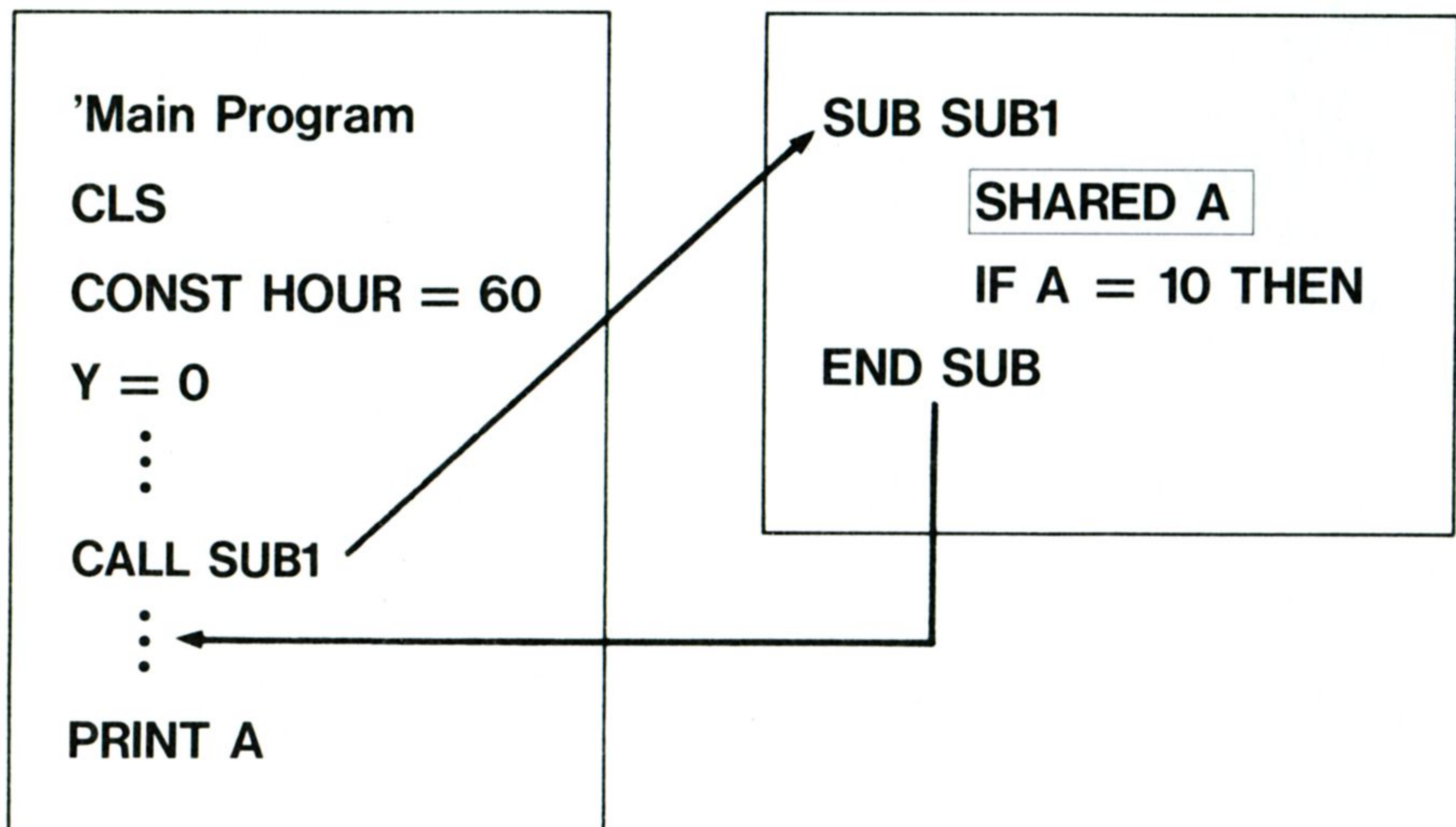
グローバル変数とは、特定の範囲内だけ意味を持つローカル変数に対し、全ての領域に対して効力を持つ変数のことをさします。

●グローバル変数の概念

〈メインルーチン〉

```
'Main Program
CLS
CONST HOUR = 60
Y = 0
  :
CALL SUB1
  :
PRINT A
```

〈SUBプロシージャ〉

```
SUB SUB1
    SHARED A
    IF A = 10 THEN
END SUB
```

□で示されている部分が変数のグローバル化のための宣言です。

グローバル変数となった変数は、サブルーチン内で変更されても、その変数の内容をメインルーチンに持ち込みます。

ただし、ここでいうグローバル変数とは「特定のサブルーチンにだけ有効」なグローバル変数であるということに注意してください。

SUBプロシージャの場合には、変数AはSUBプロシージャSUB1の中だけで変数の内容を保持していましたが、SHAREDステートメントを用いることで変数Aはメインルーチンでもその値を参照することができます。

2 グローバル変数の使い方

まず、グローバル変数にしない場合のプログラミングをしてみましょう。

●ソースリストPRG-NO46.BAS

```
DECLARE SUB SUB1 ()
CLS
A = 1
B = 2
PRINT "MAIN  A ="; A; "   B ="; B
CALL SUB1
PRINT "MAIN  A ="; A; "   B ="; B
```

●SUBプロシージャSUB1

```
SUB SUB1
    A = 3
    B = 4
    PRINT "SUB   A ="; A; "   B ="; B
END SUB
```

この場合、SUBプロシージャSUB1で変数をグローバル化していないので、例え同じ変数名でも変数A・Bは別の変数として扱われます。

実行してみましょう。

●インタプリタによる実行結果

```
MAIN   A = 1    B = 2
SUB    A = 3    B = 4
MAIN   A = 1    B = 2
```

次が、グローバル変数にした場合のプログラミングです。

●ソースリストPRG-NO47.BAS

```
DECLARE SUB SUB1 ()
CLS
A = 1
B = 2
PRINT "MAIN  A ="; A; "    B ="; B
CALL SUB1
PRINT "MAIN  A ="; A; "    B ="; B
```

●SUBプロシージャSUB1

```
SUB SUB1
   SHARED A, B ←―――この一行を追加
   A = 3
   B = 4
   PRINT "SUB   A ="; A; "    B ="; B
END SUB
```

SHAREDステートメントを使用しているので、変数A・B共にメインルーチンに影響を及ぼし、それぞれの変数の値が変わっています。

実行してみましょう。

●インタプリタによる実行結果

```
MAIN   A = 1    B = 2
SUB    A = 3    B = 4
MAIN   A = 3    B = 4 ←―――メインルーチンの値も変化している
```

実際にプログラミングを行なうようになると、SUBプロシージャやFUNCTIONプロシージャの変数が、全てローカル変数ばかりでは困ることがあります。

全てのプログラム中で使いたい変数や、ケースバイケースでグローバルにしたいときもあります。

このようなときには、SHAREDステートメントで変数をグローバル化してください。

第12章

トラッピング

トラッピングとは、プログラムの流れとは全く無関係に制御を変更することをいいます。

時として市販のアプリケーションでは、どんな状態からでもESCキーを押せばメニューが表示されるなどのように、プログラムの流れを無視したことを平気で行なっています。

実は、これらの処理のほとんどは、トラッピングで実現しています。

ここでは、こういった特定のキーに対するトラッピングから、エラーが発生したときのトラッピングを学びましょう。

12・1

イベントトラッピング

指定されたキーを押されたとき、指定の作業を行なう

1　イベントトラッピングとは

イベントトラッピングとは、プログラム中で指定したキーを押されたときに、指定の作業を行なうことをいいます。

例えば、ESCキーをイベントキーと指定し、イベントキーが押されたときの処理を「ブザーを1回ならす」とします。

このような設定がなされているときには、ESCキーは本来のキーの機能は無視され、イベントトラッピング以外には使用できなくなります。

そして、ESCキーが押されたときには、プログラムの流れに全く無関係に、ブザーを1回鳴らします。

このような流れで、イベントトラッピングが行なわれます。

イベントトラッピングは、次のような宣言を行ないます。

●●イベントトラッピングの宣言●●

- **イベント発生時の分岐先指定**
  ON KEY [キーナンバー] GOSUB [ラベル]
- **イベントに対してキーの効力を無効にする**
  KEY [キーナンバー] ON
- **イベントに対してキーの効力を無効にする**
  KEY [キーナンバー] OFF

・キーナンバー

イベントトラッピングに指定したいキーのナンバーを指定します。(注意：キーのナンバーについては、巻末の「キーコード表参照」)

・ラベル

トラッピングが発生したときの行き先を指定します。

2　イベントトラッピングの使い方

イベントトラッピングを行なうには、まず、「イベントとして指定したキーが押されたときの分岐先」を明確にするため、“ON KEY...”ステートメントを用います。

次に、そのキーが押されたときに分岐させるのか、または分岐させないのかを明確にします。この操作を“KEY ... ON/OFF”によって行ないます。

イベントトラッピングでは、ファンクションキーのように、機能選択に使うキーはあらかじめ簡単に使えるように指定してあります。

例えば、[f･1]はキーナンバー1を指定し、[f･10]はキーナンバー10を指定します。また、カーソル移動などによく使われる[←][→][↑][↓]は、キーナンバー11～14まで割り振ってあります。

このほかに、ユーザーが任意に指定することのできるキーがあります。例えば、先ほどの[ESC]キーなどは、ユーザーがイベントトラッピングの対象となるように独自に指定する必要があります。

ユーザーが独自に指定するキーは、キーナンバー15～25までです。

次に、簡単なプログラムを通して、イベントトラッピングを行なってみることにします。

●ソースリストPRG-NO48.BAS

```
CLS
ON KEY(1) GOSUB F1  ←ファンクションキー[f.1]が押されたときはラベルF1にとぶ
KEY(1) ON  ←KEY(1)を有効にする
A$ = "A"

DO
    PRINT A$[;]  ←セミコロンを最後に付けると次のPRINT文の
                   メッセージを連続して表示する
    K$ = INKEY$
    IF K$ = "*" THEN
        EXIT DO
    END IF
LOOP
END

F1:
    PRINT
    INPUT "Please data : ", A$
RETURN
```

実行してみましょう。

●インタプリタによる実行結果

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA□ ── ここで[f.1]が押された
Please data : Z
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZ□ ── ここで[f.1]が押された
Please data : Q
QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
QQQQQQ□
      └── ここで“*”が押されたので終る
```

実際は、プログラムの制御の方がキー入力よりも早いので、もっと多くの英文字が出力されていると思います。

[f・1]を押すと、メインとなる制御（ここでは文字出力）を無視して変数の値が変わっていることが分かりますね。

次に、カーソルキーを移動させるイベントトラッピングを行なってみることにします。

●ソースリストPRG-NO49.BAS

```
CLS
ON KEY(11) GOSUB CURUP ──────── ↑キー
ON KEY(14) GOSUB CURDOWN ────── ↓キー
ON KEY(12) GOSUB CURRIGHT ───── ←キー
ON KEY(13) GOSUB CURLEFT ────── →キー
KEY(11) ON
KEY(12) ON
KEY(13) ON
KEY(14) ON

PRINT "Return key is quit..."

DO                          ┌── リターンキーを検出するテクニック
    K$ = INKEY$             │
    IF K$ = CHR$(13) THEN
        EXIT DO
    END IF
    PRINT K$;
LOOP
END

CURUP:
    IF CSRLIN <= 1 THEN
        BEEP
        RETURN               ── もし現在のカーソル位置が最上段までいって
    END IF                      いた時には，ベルを1回鳴らしてメインルー
                                チンに戻る

    LOCATE CSRLIN - 1, POS(0)
RETURN                      └── カーソルの位置を1つ上にずらす

CURDOWN:
    IF CSRLIN >= 25 THEN
        BEEP
        RETURN               ── もし現在のカーソル位置が最下段までいって
    END IF                      いた時にはベルを1回鳴らしてメインルーチ
                                ンに戻る
    LOCATE CSRLIN + 1, POS(0)
RETURN                      └── カーソルの位置を1つ下にずらす
```

```
CURRIGHT:
    IF POS(0) <= 1 THEN
        BEEP
        RETURN
    END IF
    LOCATE CSRLIN, POS(0) - 1
RETURN

CURLEFT:
    IF POS(0) >= 80 THEN
        BEEP
        RETURN
    END IF
    LOCATE CSRLIN, POS(0) + 1
RETURN
```

もし現在のカーソル位置が左までいっていた時にはベルを1回鳴らしてメインルーチンに戻る

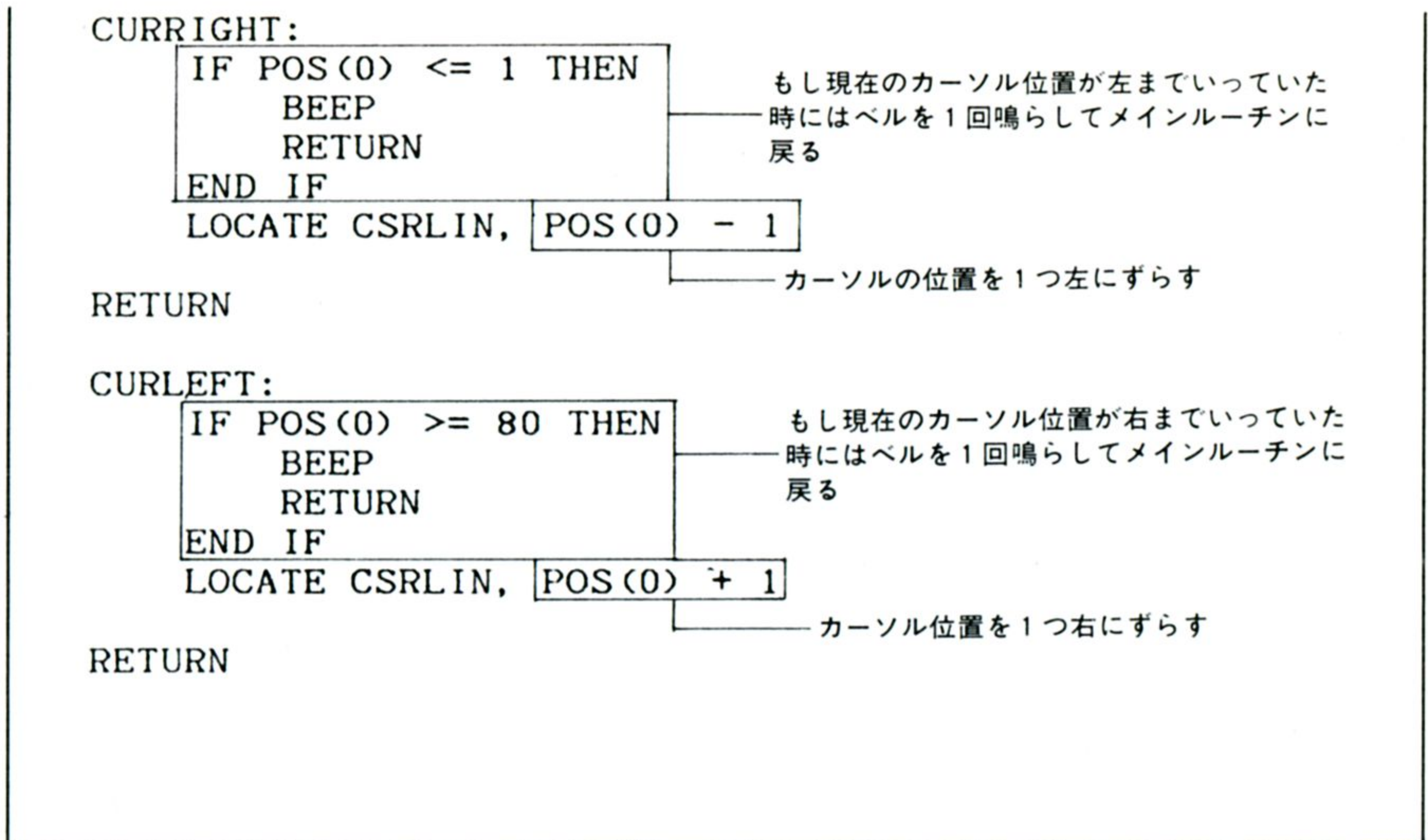
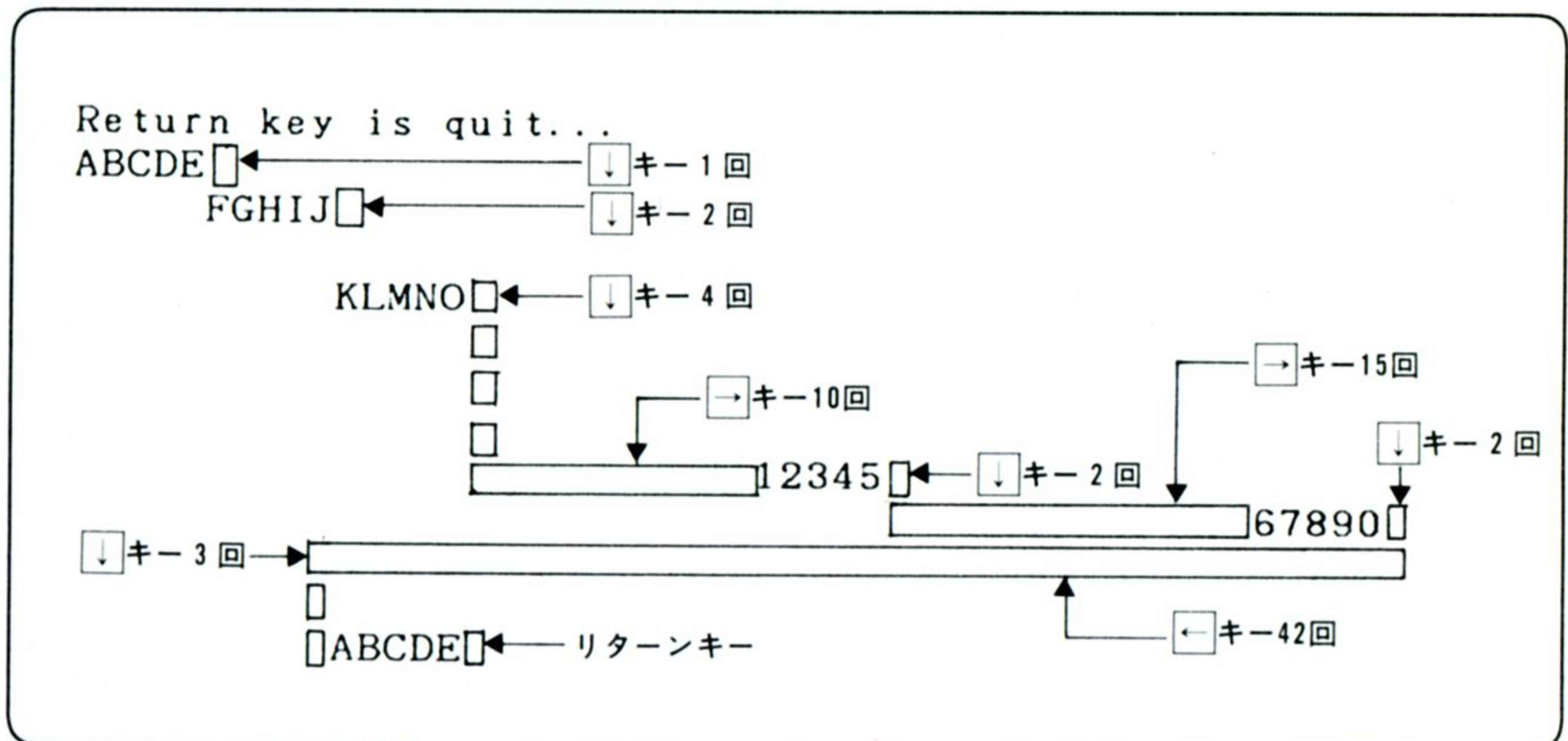
カーソルの位置を1つ左にずらす

もし現在のカーソル位置が右までいっていた時にはベルを1回鳴らしてメインルーチンに戻る

カーソル位置を1つ右にずらす

●インタプリタによる実行結果

```
Return key is quit...
ABCDE□
        FGHIJ□

                KLMNO□
                     □
                     □
                     □
                     [          ]12345□
                                      [          ]67890□
           [                                               ]
           □
           □ABCDE□
```

↓キー1回
↓キー2回
↓キー4回
→キー10回
↓キー2回
→キー15回
↓キー2回
↓キー3回
←キー42回
リターンキー

　イベントトラッピングの大まかな流れはつかんでもらえたと思います。

　しかし、この例だけでは、ESCキーを押したときなどのトラップができません。

今度はユーザー定義のキーに対してのイベントトラッピングを行なうことにします。

●ソースリストPRG-NO50.BAS

```
CLS
KEY 15, CHR$(0) + CHR$(&H3F)    ← キーの定義
ON KEY(15) GOSUB HELP           ← 定義したキーのトラップを指定
KEY(15) ON
PRINT "Return key is quit..."

DO
    K$ = INKEY$
    IF K$ = CHR$(13) THEN
        EXIT DO
    END IF
    PRINT K$;
LOOP
END

HELP:
    BEEP
    PRINT
    PRINT "Press <HELP> key."
RETURN
```

ここでは、HELPキーをイベントトラッピングとして指定しました。

HELPなども、ヘルプメッセージを表示させるためよく使われるキーですが、メッセージを表示させるためにキーチェックをたえず行なうよりも、このようにイベントトラップに指定した方が、効率的なプログラムになると思います。

ただし、注意点が2つあります。

まず第1に、上記のプログラムの場合には、CAPSキーがロック状態（押し込まれている）の場合には、HELPキーを検出しません。したがって、CAPSキーのロックを解除しないと、期待した動作は得られないことになります。

第2に、“CHR$ (0) + CHR$ (&H3F)”のように、任意のキーをイベントトラッピングに使うには、“CHR$ (0) +”のように指定のキーだけでなく、対応のキーをたす必要があります。

この2点を満たせば、正常に任意のキーのイベントトラッピングが行なえます。

実行してみましょう。

●インタプリタによる実行結果

```
Return key is quit...
abcde□◀――――――― ここでHELPキーを押す
Press <HELP> key.
1234567890□◀――――― ここでHELPキーを押す
Press <HELP> key.
-/*+=□◀――――――― ここでリターンキーを押す
```

［注意］

KEY 15, CHR$ (0) + CHR$ (&H3F)

の数値&H3Fは、16進数のHELPキーのキーコード。詳しくは、巻末の「キーコード表」を参照。

12・2

エラートラッピング

プログラム実行中にエラーが発生したときにトラッピング

1　エラートラッピングとは

エラートラッピングとは、プログラムを実行中にエラーが発生した時にトラッピングを行なうことをいいます。

例えば、「ドライブ装置にフロッピーディスクが挿入されていない」などの時態に遭遇した場合、何も対処していないときには実行中のプログラムが破壊されたり、そこまでいかなくてもMS-DOSのエラーメッセージのために画面が乱されたりします。

このようなことにならないように、あらかじめ予想されるエラーに対して、エラートラッピングの処置を施しておけば、プログラムがある程度乱暴な使われ方をしても対処できるでしょう。

エラートラッピングは、次のような宣言を行ないます。

●●エラートトラッピングの宣言●●

・分岐先の指定

ON ERROR GOSUB [ラベル]

・エラーの種類の識別

ERR関数

・エラールーチンからの復帰

エラーの発生した直後のステートメントから実行

RESUME NEXT

エラーの発生したステートメントから実行

RESUME 0

特定のラベル位置に分岐

RESUME [ラベル]

・分岐先の指定

エラートラッピングの場合には、イベントトラッピングのようにキーを指定したりすることはありません。単に、「エラーが発生した」ということしか検出しないので、宣言は簡単です。

・エラーの種類の識別

発生したエラーの種類の判別は、ERR関数によって行なわれます。

エラーが発生した場合、Quick BASICにはそのエラーがどのようなエラーなのかをコード番号で返します。

このコード番号をERR関数を使って読み出せば、おのずとエラーの種類が分かります。

・エラールーチンからの復帰

エラールーチンからの復帰はRESUMEステートメントを使用します。

エラートラッピングの場合にはイベントトラッピングと違い、トラッピングの処理が終った後の処理を選択できます。

この点はケースバイケースなので何ともいえませんが、通常は、エラーの発生した直後の行から復帰させます。

誰でもエラーには遭遇しますし、ましてや、不特定多数の人が使用するプログラムを書く場合には、もやはエラートラッピングは必須の項目となるでしょう。

2　エラートラッピングの使い方

では、実際にプログラミングを行なって、エラートラッピングの使い方を覚えて下さい。

●ソースリストPRG-NO51.BAS

```
ON ERROR GOTO FERROR          ←エラートラッピングの設定
CLS

DO
    INPUT "Please FILE name : ", FILENAME$
    IF FILENAME$ = "END" OR FILENAME$ = "end" THEN
        EXIT DO
    END IF                    ←起動するプログラム名を入力
    CHAIN FILENAME$           ←外部のプログラムを起動するステートメント
LOOP
END

FERROR:
    CLS
    BEEP
    COLOR 3
    PRINT "<<< FILE ERROR >>>   ";
    SELECT CASE ERR           ←ERR関数を使ってエラーコードをそのまま取得
        CASE 53
            PRINT "File not found..."     ←指定のファイルがなかった場合のエラーメッセージ
        CASE 71
            PRINT "Not ready drive..."    ←ドライブの準備がなされていなかった場合のエラーメッセージ
        CASE ELSE
            PRINT "Else <ERROR>..."       ←その他のエラーの場合のエラーメッセージ
    END SELECT
    COLOR 7

RESUME NEXT                   ←エラー発生行の直後に復帰
```

◉インタプリタによる実行結果

```
Please FILE name : ABCD              ←めちゃくちゃなファイル名を入力

<<< FILE ERROR >>>  File not found...  ←ファイルがない時の
                                         メッセージ
Please FILE name : C:¥PRG-NO1        ←ドライブがロックされていないディスク装置を指定

<<< FILE ERROR >>>  Not ready drive...  ←ドライブが準備されていない時のメッセージ

Please FILE name : PRG-NO1
QUICK BASIC
```

プログラムのポイント

■CHAINステートメントで外部プログラムを呼び出す

このプログラムは、CHAINステートメントを使って、外部プログラムを呼び出して実行させています。

このようなことを行なわせると「指定のファイルがない」や「ドライブの準備ができていない」などのエラーによく遭遇するものです。

そこでここでも、ファイルがなかったときとドライブの準備がされていなかった時のエラーメッセージを出力させています。

■CHAINステートメントの注意点

CHAINステートメントは、外部プログラムを呼びだして実行することを目的としたステートメントですが、呼び出す側の（ここではPRG-NO51.BAS）プログラムの動作環境により、呼び出し可能なプログラムと不可能なプログラムがあります。

CHAINステートメントを用いて呼びだし可能なプログラムは、呼び出す側のプログラムのモードにより左右されます。

つまり、呼び出す側のプログラムがQuick BASICのエディタ内からの実行を行なった場合には、同じ＊.BAS形式のプログラムしか呼び出せません。

また、コンパイルを終了させた＊.EXEモデルから他のプログラムを呼び出す場合には、＊.EXEなどの実行形式のプログラムしか呼び出せません。

このように、呼び出すプログラムのモードにより呼び出せるプログラムに制限があります。

■Quick BASIC環境内から呼び出す

```
CHAIN PRG...
```

この場合には、コンパイルする前のプログラムが実行可能

■コンパイル後のプログラムから呼び出す

```
CHAIN CHKDSK
```

この場合には、＊.EXEモデルなどのプログラムが実行可能

不注意にモードの違うプログラムを呼び出すと、暴走することがあるので十分注意してください。

第 13 章

ファイル操作

ファイル操作は、コンピュータの処理では必須といっていいでしょう。
データを蓄えたり、またそのデータを元に解析を行なうなど、コンピュータで処理を行なうには、ファイル処理はなくてはなりません。
ここでは、ファイル操作の基礎から簡単なデータベースの作り方までを解説します。

13・1

データファイルの種類

シーケンシャルファイルとランダムファイル

1　データファイルとは

今まで私たちが作成してきたプログラムは、全て＊.BASもしくは＊.EXEというファイル名で保存されています。

これらのファイルは、単純に見て「実行形式」のプログラムであり、データを蓄えておくファイルではありません。

ちょっとしたデータならば実行形式のプログラムに入れておくこともできますが、通常はデータ形式のファイルと実行形式のファイルを分けておきます。

Quick BASICのファイル名は、＊.BASもしくは＊.EXEと決っていましたが、データファイルはどんなファイル名でも構いません。

ただ、「データである」ということが明確なファイル名の方が管理がしやすいので、多くのプログラマは

＊.DAT　　＊.DEF　　＊.ID

などのような拡張子を付けて保存します。

●実行形式のファイル

```
CLS
CONST HOUR = 60
CONST MIN = 60
      ⋮
FOR X = 1 TO 100
    LOCATE 3,10
        ⋮
```
―プログラム

●データファイル

```
"SUZUKI YUUJI",1,30,"A","143"
"MATUDA SHINSUKE",1,26,"A","333"
                    ⋮
"SATOU YUUKO",0,22,"B","156"
                    ⋮
```
―データ

2　シーケンシャルファイルとランダムファイル

実行形式のファイルとデータファイルの区別は分かりました。

では、データファイルとは、どんな形式でデータを蓄えるのでしょうか。

Quick BASICでは、シーケンシャルファイルとランダムファイルという形式の二通りの方式でファイルにデータを蓄えることができます。

①シーケンシャルファイル

シーケンシャルファイルとは、データを積み重ねていく方式でデータを蓄えます。

また、データとデータの間には隙間を作らず、どんどん詰め込んでデータを蓄積させます。

文字型のデータは「""」で囲まれ、数値型のデータはそのまま書き込まれます。また、データとデータの隙間は「,」で区切ります。先ほどの例は、シーケンシャルファイルの中身を表したものです。

シーケンシャルファイルのメリットは、一つ一つのデータの長さを決めておく必要がないこと、テキストという方法で保存してあるので、TYPEコマンドなどで簡単に表示修正ができることなどが挙げられます。また、プログラムを作成する上でも保守が比較的簡単に行なえます。

```
"SUZUKI YUUJI",1,30,"A","143"
"MATUDA SHINSUKE",1,26,"A","333"
          :
          :
```

②ランダムファイル

ランダムファイルは、データを積み重ねていく点は同じですが、あらかじめデータの長さを決めておく必要があります。したがって、長さが不特定なデータを大量に扱う場合には不向きです。また、データはバイナリという機械にだけ分かる方法で保存してあるので、TYPEコマンドなどを使っても内容を理解することはできません。

しかし、シーケンシャルファイルでは表示する/しないに関わらずデータを全て読まないと目的のデータを表示することができませんが、ランダムファイルではレコード番号というのを呼び出すだけで目的のデータを表示できます。

```
SUZUKI YUUJI        MATUDA SHINSUKE
1                   1
30                  26
A                   A
143                 333
```

一般に広く使われているのは、ランダムファイルの方ですが、データを扱えるまでに覚えることが多いので、ここではシーケンシャルファイルを例にとってファイル操作を行なうことにします。

シーケンシャルファイルが簡単だからといってバカにしてはいけません。

最近では、不定長のデータを扱うデータベースが増えていますが、これらのデータベースはほとんどシーケンシャルファイルの形式です。

13・2

ファイルを操作する

ファイルのオープンとデータの書き込み、読み出し

扱うファイルなどの大まかな流れをつかみましたので、次に実際に操作を行ないながら、ファイル操作の感覚をつかんでいくことにしましょう。

1 ファイルのオープン

シーケンシャルファイルに限らず、データファイルを「使う」場合には、オープンということを行ないます。

ファイルのオープンとは、「データファイルを使えるように準備する」と考えてください。

ファイルを使えるように準備するといっても、なにもデータを書き込んでいないファイルを新しく作る場合や、すでにあるファイルにデータを追加するなど、作業はファイルの状況によって左右されます。

このことをふまえて、ファイルのオープンを行なってみましょう。

データファイルをオープンするといってもいくつかの決りがあります。ここでは、ファイルをオープンする際の規則を理解します。

●●新規データファイルのオープン●●

- **データファイルがすでに存在し、そのファイルの内容を全て無効にしてファイルをオープン**
  OPEN "[ファイル名]" FOR OUTPUT AS #1
- **データファイルがすでに存在した場合にはデータを追加するようにオープンデータファイルが存在しなければ新規にデータファイルを作成**
  OPEN "[ファイル名]" FOR APPEND AS #1
- **データファイルをデータが読み込めるようにオープン**
  OPEN "[ファイル名]" FOR INPUT AS #1

OPENステートメントにより、ファイルのオープンは行なわれます。

OPENステートメントにはいくつかのオプションがあります。

①ファイル名

OPEN "[ファイル名]" FOR OUTPUT AS #1

ここには、オープンすべきファイル名を書き込みます。FORMATコマンドなどを実行するときのように、拡張子を省略してはなりません。

次の例は、"SAMPLE.DAT" というファイルをオープンする例です。

［例］OPEN "SAMPLE.DAT FOR OUTPUT AS #1

②ファイルのモード

OPEN "[ファイル名]" FOR OUTPUT AS #1

オープンするファイルのモードを決定します。

新規にファイルをオープンする場合には、通常OUTPUTをモードとしますが、OUTPUTを使用してファイルをオープンすると、新規にファイルを作成しますが、指定したファイルがディスク上に存在するとそのファイルの内容は全て破棄されますので注意が必要です。

また、新規にファイルをオープンするには、APPENDをモードとしても使えます。

指定のファイルがディスク上に存在しなければ新規にファイルを作成し、存在すればそのファイルにデータを追加するモードでオープンします。

また、INPUTをモードとして指定すると、データを読み取るようにしてファイルをオープンします。

いずれにしろ、実際にプログラム中でデータファイルを扱う場合には、ファイルが存在するか/しないかの検査をしてからファイルをオープンした方がいいでしょう。

次の例は、OUTPUTモードを使用して“SAMPLE.DAT”というファイルをオープンした例です。

［例］OPEN "SAMPLE.DAT" FOR OUTPUT AS #1

③ファイル番号

OPEN "[ファイル名]" FOR APPEND AS #1

Quick BASICでは、いったん読み込んだファイルは全て番号で操作されます。オープンするときだけファイル名を指定します。

いったん指定さえすれば、以降“#1”などのように扱われます。

次の例は、“SAMPLE.DAT”というファイルに“#3”という番号を割り振っています。

［例］OPEN "SAMPLE.DAT" FOR OUTPUT AS #3

ファイルをオープンしデータを書き込んだ後には、かならずクローズという作業を行ないます。

この作業を行なわずにプログラムを不注意に終了させると、せっかく入力したデータなどを失うことがあるのでこの点も注意してください。

2　ファイルオープンのプログラミング

それでは、実際に、ファイルをオープンするプログラムにチャレンジしてみましょう。

●ソースリストPRG-NO52.BAS

```
ON ERROR GOTO FERR ←エラー発生時はFERRにとぶ
CLS
FILES "TEST.DAT" ←TEST.DATというファイルがあるか検査
PRINT "File open test."
OPEN "TEST.DAT" FOR OUTPUT AS #1 ←TEST.DATというファイルをオープン
CLOSE #1←ファイル番号#1をクローズ
FILES "TEST.DAT" ←TEST.DATというファイルがあるか検査
END

                  ┌ファイルが存在しない時のエラーコード
FERR:             ↓
    IF ERR = [53] THEN
        PRINT "File not found..."
    END IF
   DO
   LOOP UNTIL INKEY$ <> ""
RESUME NEXT
```

このプログラムは、OPENステートメントを使用して、“TEST.DAT”というファイルをディスク上に作成している例です。

単にTEST.DATを作成しているだけですが、ファイルが存在するかのチェックなども行なっているので、応用次第では使えるかと思います。

●インタプリタによる実行結果

```
B:¥Q-BASIC←ディレクトリを見る
[File not found...]←TEST.DATというファイルは存在しない
Hitany key
B:¥Q-BASIC←再度ディレクトリを見る
TEST    .DAT ←TEST.DATが作られていた
 3248128 バイトが未使用です
```

さて、実行結果を見ると最初のファイル検査（FILESステートメント）ではTEST.DATというファイルがなかったため、エラーメッセージを出力されました。

次にOPENステートメントを使用してファイルを新規に作成、そしてCLOSEステートメントによってファイルをクローズしています。

この作業を行なえば、ファイルは作成されているはずです。

最後に、もう一度ファイルが存在するかチェックしています。

3 データを書き込む

ファイルのオープンが分かりましたが、これだけでは実際になんの役にも立ちません。データを書き込んでこそファイル操作は活用できるのです。

そこでここでは、データを書き込む方法を理解します。

データを書き込むには、次の形式を使用します。

●●オープンしたファイルにデータを書き込む●●

WRITE #ファイル番号

ファイル番号とは、ファイルをオープンしたときに付けた番号です。

また、ファイルにデータを書き込む場合、ファイルモードに気を付けてください。

重ねていいますが、OUTPUTでファイルをオープンすると、ディスク上にファイルが存在した場合、そのファイルの内容を全て破棄しますので、データの追加をすることにはなりません。

OUTPUTでファイルをオープンする場合は、新規にファイルを作成する場合にだけ使用します。

既存のファイルにデータを追加する場合には、APPENDモードを使用してファイルをオープンします。

では、実際にデータを書き込むためのプログラミングをしてみましょう。

◉ソースリストPRG-NO53.BAS

```
CLS
OPEN "PEOPLE.DAT" FOR APPEND AS #1
DO
    CLS
     INPUT "NAME : ", PNAME$
     INPUT "AGE  : ", PAGE
     INPUT "SEX  : ", PSEX$
     WRITE #1, PNAME$, PAGE, PSEX$
     PRINT
     PRINT "Write data..."
     PRINT
     INPUT "Continue ? [ Y/N ] ", YN$
     IF YN$ = "N" OR YN$ = "n" THEN
          CLOSE #1
          CLS
          PRINT "End programs"
          EXIT DO
     END IF
LOOP
END
```

追加モードでファイルをオープン（APPEND）
それぞれの変数にデータを入れておく（PNAME$, PAGE, PSEX$）
入れておいた変数をファイル番号#1に書きこむ（PNAME$, PAGE, PSEX$）
ファイル番号#1をクローズしてから終了（CLOSE #1）

◉インタプリタによる実行結果

```
NAME : WADA TOSHIKATU
AGE  : 21
SEX  : MEN

Write data...

Continue ? [Y/N ] Y

NAME : MANITA CHIHARU
AGE  : 23
SEX  : WOMAN

Write data...

Continue ? [Y/N ] Y
```

```
NAME : ISHII BARG
AGE  : 21
SEX  : MEN

Write data...

Continue ? [Y/N ] N

End programs
```

このデータがファイルとして保存されたかどうか、MS-DOSで確認してみます。

●MS-DOSによるサンプルオペレーション

```
A>TYPE PEOPLE.DAT ←ファイルに保存されたか確認する

"WADA TOSHIKATU",21,"MEN"      ┐
"MANITA CHIHARU",23,"WOMEN"    ├たしかに保存された
"ISHII BARG",21,"MEN"          ┘

A>
```

ここでは、追加モードのAPPENDを使って新規にファイルを作成しました。この方法を用いれば、誤操作でファイル内容を消してしまうといったことがありません。

次にこのプログラムを実行すると、上記のデータの次に新しいデータを書き込みます。

4　データを読み出す

ファイルにデータを登録する方法の次は、書き込んであるデータを読み出してみます。

データを読み出すには、INPUTモードを使ってファイルをオープンすればいいだけです。

●●ファイルのデータを読み出す●●

・ファイルを読み出し用としてオープン

OPEN "[ファイル名]" FOR INPUT AS #ファイル番号

・変数に読み出したデータを代入

INPUT #ファイル番号，変数名，変数名，....

ちょうど、WRITEステートメントの逆を行なうわけです。

ここで注意したいのが、変数の並び方です。WRITEステートメントを使用してデータを書き込んだとき、変数の並び方は特に注意する必要がありませんでしたが、データを読み出す場合には変数の並び方に注意しなくてはなりません。つまり、「書き込んだ順番通りに読み出す」必要があります。

例えば、一番最初のデータは文字型なのに数値型で読みだそうとするとエラーとなります。変数の方が違うので代入ができません。

また、先ほどの例では１つのデータの塊として、名前・年齢・性別と３つのデータを書き込みましたが、INPUTステートメントで書き込んだ数だけ扱ってやらないとエラーとなりますので注意してください。

●ソースリストPRG-NO54.BAS

```
CLS
OPEN "PEOPLE.DAT" FOR INPUT AS #1        ←ファイルを読み出しモードでオープン
DO UNTIL EOF(1)        ←ファイルが終りに達したことを得る関数

    INPUT #1, PNAME$, PAGE, PSEX$        ←それぞれの内容を表示

    PRINT "NAME : "; PNAME$        ←変数の内容を表示
    PRINT "AGE  : "; PAGE
    PRINT "SEX  : "; PSEX$
    PRINT
LOOP
PRINT "End data..."
CLOSE #1
END
```

●インタプリタによる実行結果

```
NAME : WADA TOSHIKATU
AGE  :  21
SEX  : MEN

NAME : MANITA CHIHARU
AGE  :  23
SEX  : WOMAN

NAME : ISHII BARG
AGE  :  21
SEX  : MEN
End data...
```

ちゃんとこのようになりましたか。

データを読み出す場合には、「ファイルの最後まで達した」ということを検出しないとなりません。これを怠ると「ファイルが終りに達しました（ERR=62）」というメッセージが出力されます。

このメッセージは､「ファイルの終りにきているのにまだ読み込もうとした」ときに出力されるメッセージです。よって、EOF関数を使用して、ファイルの終りにきたら読み込みをしてはなりません。

これで一通りのことを行ないました。

データファイルの新規作成・データの追加・データの表示と、これだけ知っていれば簡単なデータベースぐらいは作成できます。

ただし、既存のデータに変更を加えることはできません。

そこで、次に簡単なデータベースを作成すると共に、データを修正するテクニックを紹介します。

13・3

データベースを作ろう

データを変更できる電話帳

ここでは、今まで学んだ知識で電話帳を作ります。

本来ならばプログラムの仕様書を作成してからプログラムを書きますが、あまり長くないので、プログラムのリストを先にお見せすることにします。

◉ソースリストPRG-NO55.BAS

```
DECLARE SUB CARA ()                 ←サブルーチン群
DECLARE SUB MAINTITLE ()
DECLARE SUB DISP (TITLE$)
DECLARE SUB APPDATA ()
DECLARE SUB CNVDATA ()
DECLARE SUB DSPDATA ()
                                    "0"が押されるまで回り続ける
DO
    CALL MAINTITLE                  ←メインメニュー表示

    DO
        LOCATE 17, 46               ←入力キーのチェック
        K$ = INKEY$
        PRINT K$
    LOOP UNTIL K$ <> ""
                                    入力したキーにより
                                    それぞれの作業開始
    SELECT CASE K$

        CASE "0"                    ←終了
                                      メインループを抜ける
            EXIT DO
        CASE "1"                    ←データ追加
                                      APPDATA呼び出し
            CALL APPDATA
```

```
CASE "2"                    ←―データ修正
                              CNVDATA呼び出し
    CALL CNVDATA
CASE "3"                    ←―データ表示
                              DSPDATA呼び出し
    CALL DSPDATA
CASE ELSE                   ←―違法な入力の場合にはループに戻る
    BEEP
END SELECT
LOOP

CLS
PRINT "End programs"
END
```

●SUBプロシージャAPPDATA　データ追加用

```
SUB APPDATA
    OPEN "TEL.DAT" FOR APPEND AS #1
    CALL DISP("Append data")  ←―SUBプロシージャDISPでサブタイトルの表示
    DO
        LOCATE 3, 21: INPUT "", TELNAME$
        LOCATE 4, 21: INPUT "", TEL$
        WRITE #1, TELNAME$, TEL$
        LOCATE 7, 1: PRINT "Continue ? [ Anykey = Yes, N = No ] "
        DO
            K$ = INKEY$
        LOOP UNTIL K$ <> ""
        IF K$ = "N" OR K$ = "n" THEN
            EXIT DO
        END IF
        CALL CARA
    LOOP
    CLOSE #1
END SUB
```

●SUBプロシージャCARA　入力領域の消去

```
SUB CARA                                  空白を指定数出力する関数
    BEEP                                  ( )内が出力する空白の数
    LOCATE 3, 21: PRINT SPACE$(30)
    LOCATE 4, 21: PRINT SPACE$(30)
    LOCATE 7, 1: PRINT SPACE$(50)
END SUB
```

●SUBプロシージャCNVDATA　データの修正

```
SUB CNVDATA
    OPEN "TEL.DAT" FOR INPUT AS #1
    OPEN "TEMP.DAT" FOR OUTPUT AS #2     ←作業用のファイルをオープ
    CALL DISP("Convert data")              ンOUTPUTモードに注目

    DIM TELNAME$(1000)    ←作業用の配列を確保
    DIM TEL$(1000)          ここでは1001個の領域を確保

    DO                                     ┌ファイルが終りにきたら終了
        IF EOF(1) THEN
            LOCATE 7, 1: PRINT "File end... Hit anykey !"
            DO
            LOOP UNTIL INKEY$ <> ""
        EXIT DO
        END IF

        COUNT = COUNT + 1
        INPUT #1, TELNAME$(COUNT), TEL$(COUNT)
        LOCATE 3, 21: PRINT TELNAME$(COUNT)
        LOCATE 4, 21: PRINT TEL$(COUNT)
        LOCATE 7, 1: PRINT "Are you sare ? [ Y = Yes, N = No ] "
                                            └データの表示。表示をする際の変数に注目
        DO
            K$ = INKEY$            ┌修正がない場合には表示したデータをそのまま作業用の
            SELECT CASE K$         │ファイルに書き込む
                CASE "N", "n"
                    CALL CARA
                    WRITE #2, TELNAME$(COUNT), TEL$(COUNT)
入力領域消却        EXIT DO

                CASE "Y", "y"
                    CALL CARA
                    LOCATE 7, 1: PRINT "Please input data"
                    LOCATE 3, 21: INPUT "", TELNAME$(COUNT)
                    LOCATE 4, 21: INPUT "", TEL$(COUNT)
                    WRITE #2, TELNAME$(COUNT), TEL$(COUNT)

                    EXIT DO
            END SELECT     └修正がある場合には，データを修正してそのまま作業用
        LOOP                 のファイルに書き込む
    LOOP
    CLOSE #1, #2    ←オープンしていたファイルをクローズ

    KILL "TEL.DAT"                   ←今までデータを書き込んでいたファイルを
    NAME "TEMP.DAT" AS "TEL.DAT"       KILLステートメントで削除。NAMEステート
END SUB                                メントで作業用のファイルをTEL.DATとす
                                       る。
```

●SUBプロシージャDISP　サブタイトルの表示

```
SUB DISP (TITLE$)
    CLS
    PRINT "----------- <<< "; TITLE$; " >>> -----------"
    PRINT
    PRINT "   Name              : "
    PRINT "   Telepone number :"
    PRINT
    PRINT "----------------------------------------------"
END SUB
```

●SUBプロシージャDSPDATA　データの表示

```
SUB DSPDATA
    OPEN "TEL.DAT" FOR INPUT AS #1
    CALL DISP("Print data")
    DO UNTIL EOF(1)
        INPUT #1, TELNAME$, TEL$
        LOCATE 3, 21: PRINT TELNAME$
        LOCATE 4, 21: PRINT TEL$
        LOCATE 7, 1: PRINT "Continue ? [ Anykey = Yes, N = No ] "
        DO
            K$ = INKEY$
        LOOP UNTIL K$ <> ""
        IF K$ = "N" OR K$ = "n" THEN
            EXIT DO
        END IF
        BEEP
        LOCATE 3, 21: PRINT SPACE$(30)
        LOCATE 4, 21: PRINT SPACE$(30)
        LOCATE 7, 1: PRINT SPACE$(50)
    LOOP
    CLOSE #1
    DO
        LOCATE 7, 1: PRINT "File end...   Hit anykey !"
    LOOP UNTIL INKEY$ <> ""
END SUB
```

●SUBプロシージャMAINTITLE　メインメニュー表示

```
SUB MAINTITLE
    CLS
    LOCATE  3, 15: PRINT "---------- <<<TELEPHONE LIST >>> ----------"
    LOCATE  6, 28: PRINT "Append data   [ 1 ]"
    LOCATE  8, 28: PRINT "Convert data  [ 2 ]"
    LOCATE 10, 28: PRINT "Print data    [ 3 ]"
    LOCATE 12, 28: PRINT "END           [ 0 ]"
    LOCATE 15, 15: PRINT "--------------------------------------------"
    LOCATE 17, 30: PRINT "Please number [   ]"
END SUB
```

やや長くなってしまいましたが、今まで学んだことを忠実にプログラムに再現しただけなので、それほど難しくないと思います。

一度解説したことは分かると思いますので、重要なところだけ解説します。

もともとシーケンシャルファイルには、「データを修正する」とか「データを削除する」などの概念はありません。

データをどんどん蓄積する機能には優れていますが、残念ながら応用力がないのです。

そこで、修正できないのならこちらで対応する必要があります。本リストでは、“CNVDATA”というSUBプロシージャがデータの修正のためのルーチンです。

まず最初に、作業用のデータファイルを作成します。この時、OUTPUTモードでかならず新規に作成するのがミソです。この点は後で分かります。

次に配列を宣言します。ここでは、1000個分の配列を宣言して、それらの変数にTEL.DATのデータを全てほうり込みます。

さて、データを修正しない場合には配列に格納したデータをそのまま作業用のファイル（WRITEステートメントを使って）TEMP.DATに書き込みます。また、修正がある場合には、配列にほうり込んだデータを無効にして新しいデータを書き込みます。そして、作業用のファイルに書き込みます。

こうして、全てのデータを参照し終れば修正の作業は終りです。

最後に、元のデータ、TEL.DATをKILLステートメントで削除、そして今まで作業用に使っていたTEMP.DATをTEL.DATに名前を変更します。

新しいファイルを作ることにより、「データを修正する」という作業に置き換えているのです。

さて、本当は配列などを使わないで、TEL.DATの内容を読み込んだ変数をダイレクトに修正し、それを作業用ファイルに書き込んでもいいのですが、配列を使うことにより、様々な可能性が出てきます。

第一に、全てのデータは配列の添字を見ることにより、番号で管理できます。したがって、検索なども簡単にできるようになります。

また、この応用によりデータの削除にも利用できるでしょう。
　やや高度な内容になってしまいましたが、がんばればきっと理解できると思います。分からなかったらもう一度リストを見直し、また改造してファイル管理を理解してください。
　実行してみましょう。

●インタプリタによる実行結果

・メインメニュー

```
---------- <<<TELEPHONE LIST >>> ----------

            Append data  [ 1 ]

            Convert data [ 2 ]

            Print data   [ 3 ]

            END          [ 0 ]

--------------------------------------------

              Please number [   ]
```

・"1" を入力 (データ追加)

```
----------- <<< Append data >>> -----------

  Name                 : MATUDA TOSHIHIKO
  Telepone number      : 03-123-4567

-------------------------------------------

Continue ? [ Anykey = Yes, N = No ]
----------- <<< Append data >>> -----------

  Name                 : TATUKAWA REIKO
  Telepone number      : 052-987-6543

-------------------------------------------

Continue ? [ Anykey = Yes, N = No ] N
```

"N" 以外のキーを押すとさらにデータが追加できる

"N" を押したので終る。メインメニューに戻る

・"2" を入力 (データの修正)

```
----------- <<< Convert data >>> -----------

  Name                 : MATUDA TOSHIHIKO
  Telepone number      : 03-123-4567

--------------------------------------------

   Are you sare ? [ Y = Yes, N = No ] N
----------- <<< Convert data >>> -----------

  Name                 : TATUKAWA REIKO
  Telepone number      : 052-987-6543

--------------------------------------------

Are you sare ? [ Y = Yes, N = No ] Y
----------- <<< Convert data >>> -----------

  Name                 : TATUKAWA REIKO
  Telepone number      : 052-876-5432

--------------------------------------------

Please input data
```

"N" を押すと次のデータが表示される

"Y" を押したのでデータを修正

新しいデータを入力する

・"3" を入力　データの表示

```
----------- <<< Print data >>> -----------

  Name               : MATUDA TOSHIHIKO
  Telepone number : 03-123-4567

----------------------------------------------

Continue ? [ Anykey = Yes, N = No ] Y ←── "Y" を入力したので次のデータが表示される

----------- <<< Print data >>> -----------

  Name               : TATUKAWA REIKO
  Telepone number : 052-876-5432 ←── ちゃんと修正されている

----------------------------------------------

Continue ? [ Anykey = Yes, N = No ] N ←── "N" を入力したので終了
```

・"0" を入力（プログラムの終了）

```
End programs
```

第14章

グラフィックス

Quick BASICには、大変強力で、しかも簡単なグラフィックス専用のステートメントが豊富にあります。

今まで収得した技術だけでも、むろん素晴らしいプログラムが書けると思いますが、グラフィックを使えば、もっと美しく、また説得力のあるプログラムが書けます。

グラフィックを操作するのは、案外と面倒なことが付き物ですが、ここでは、簡単なLINEステートメントを使ってBOXなどを描いてみることにします。

また、最後にはLINEステートメントを応用し、アニメーションを行なってみることにしましょう。

14・1

モードを知る

テキスト画面とグラフィック画面

1　グラフィック出力ができるところ

今まで私達が文字を出力していた画面をテキスト画面といいます。

キーボードから文字を入力したり、また、変数などの内容を表示したりする画面です。この画面には斜めの線を引いたりすることができません。

不特定な線を引いたり、ある範囲を特定の色で塗りつぶしたりするには、グラフィック画面と呼ばれる画面に出力しなくてはなりません。

2　テキスト画面とグラフィック画面

①テキスト画面

テキスト画面は、透明なセロハンに似ています。この画面に出力すると後ろにある画面が透けるので、白い文字を書けば（出力すれば）クッキリと浮き上がって見えます。また、赤い文字を書いても同じようにクッキリと見えます。

しかし、黒い文字を書くとどうなるでしょう。後ろの画面が黒なので文字にはなりません。カメレオンが保護色を使うように、文字は後ろの画面に溶け込んでしまいます。

②グラフィック画面

グラフィック画面は、テキスト画面の下にあります。Quick BASICを起動した直後では、黒になっています。

ここの画面には、自由に線を引いたり、また円を描いたりすることができます。場合によっては、ある色で塗りつぶすこともできます。先ほどテキスト画面はセロハンに似ていると述べましたが、グラフィック画面に線や円を描く、塗りつぶすなどの作業を行なってもテキスト画面には影響ありません。

つまり、出力するデータを扱うところが、別の領域に確保されているのです。

こうしてそれらのデータは同時に画面に出力されるので、あたかも同じ画面にあるように見えるだけです。したがって、画面を見る限りでは一つの画面のように見えますが、テキスト画面の文字をグラフィック画面に取り込んだり（不可能ではない）グラフィック画面の絵をテキスト画面に出力することはできません。

ただし、いくらおたがいに干渉しあわないようなところにデータを確保しているからといって、黒い画面に黒で文字を書いても見えませんね。

これがテキスト画面とグラフィック画面の概念です。

14•2

グラフィックを使って絵を描こう

LINEステートメントの使い方

1　グラフィック画面の範囲

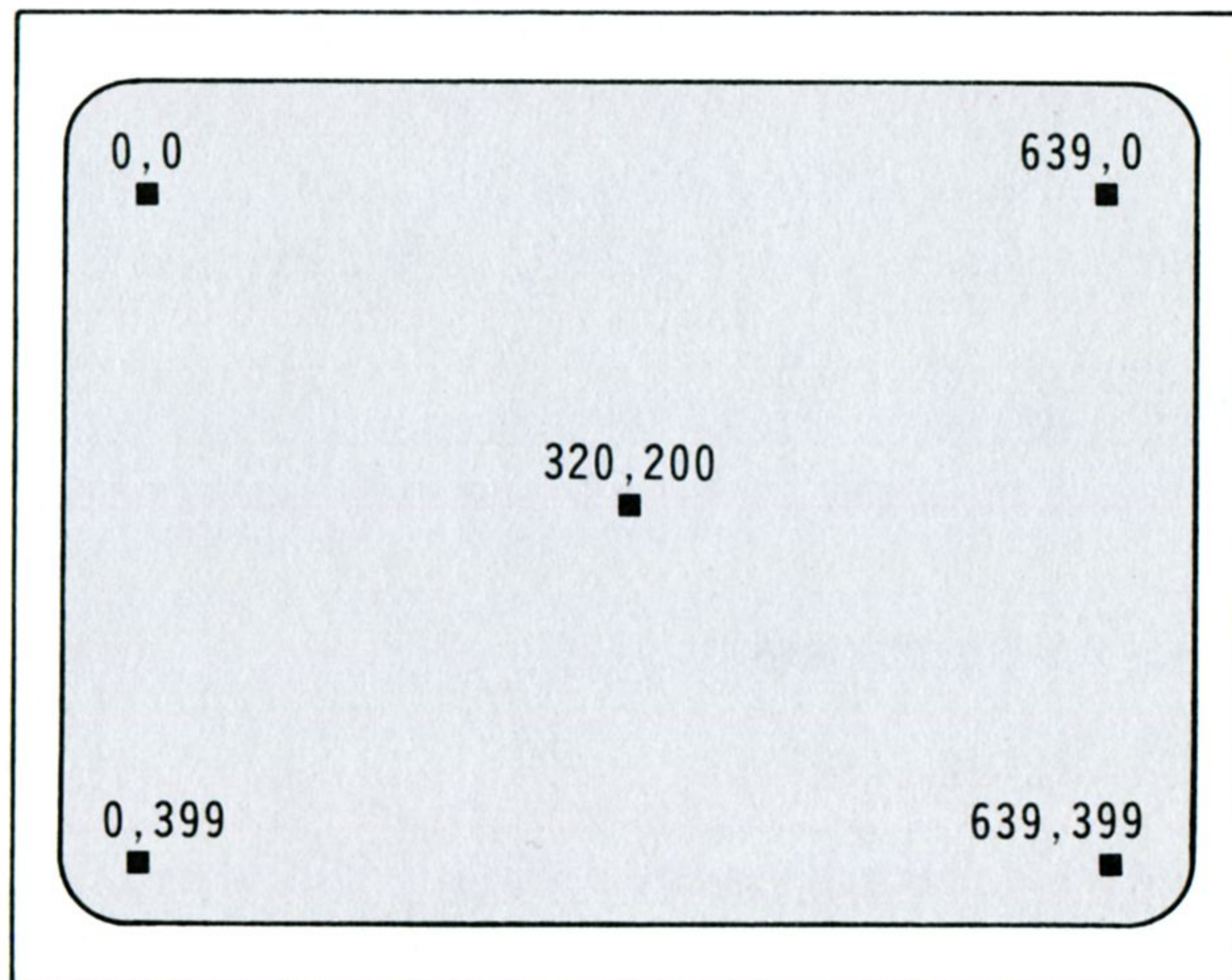
グラフィック画面には、テキスト画面のように範囲があります。

LOCATE文では、画面の左上が1,1、右下が25,80です。これがテキスト画面に出力できる範囲ですね。

では、グラフィック画面はどうなっているのでしょうか。次で詳しく解説しますが、グラフィック画面は全て点で操作するため、その範囲はテキスト画面とは比べ物にならないほど細かく分かれています。

画面の左上は0,0となり、右下は639,399となります。

◉グラフィック画面の範囲

0,0
639,0
320,200
0,399
639,399

このように、画面の中心は320,200と現されます。また、左上や右下もこのように表現されます。

縦方向は0～399まで、横方向は0～639までとなっています。これらのことをピクセルとかドットとかいう呼び方をしています。つまり、グラフィック画面の最小単位は1ピクセルもしくは1ドットということになります。

これらの表現方法は、グラフィックを出力する場合にはかならず出てくるので、覚えてください。

2　図形を描く

コンピュータのグラフィックは、全て点で表されることが分かりました。

テキスト画面に表示する文字なども点を一つの集合にしたものですが、それを操作することはできません。しかし、グラフィック画面ではそれらの点を自由に組み合せたりすることができます。

点を一直線に並べることで線を描き、丸く並べることで円を描きます。また、一定の色の点で埋め尽くせば塗りつぶすことになります。

このように、グラフィックは全て点によって表現しています。

ただし、点だけで操作するのは大変です。

そこで、線を書いたり円を描いたりするのに便利なステートメントがQuick BASICには用意されています。

これらのステートメントを使って、実際に絵を描いてみましょう。

線を引く

線を引くには、LINEステートメントを使います。

●●LINEステートメントの書式●●

```
LINE　(X1-Y1) - (X2-Y2),[色]
```

X1・Y1は、線を描く始点です。また、X2・Y2は線を描く終点です。始点と終点は、縦方向、横方向共に指定します。

やや面倒ですが、変数が使えるので、大量の線を引くときにはFOR文などを用いてループにかければ簡単に線が引けます。

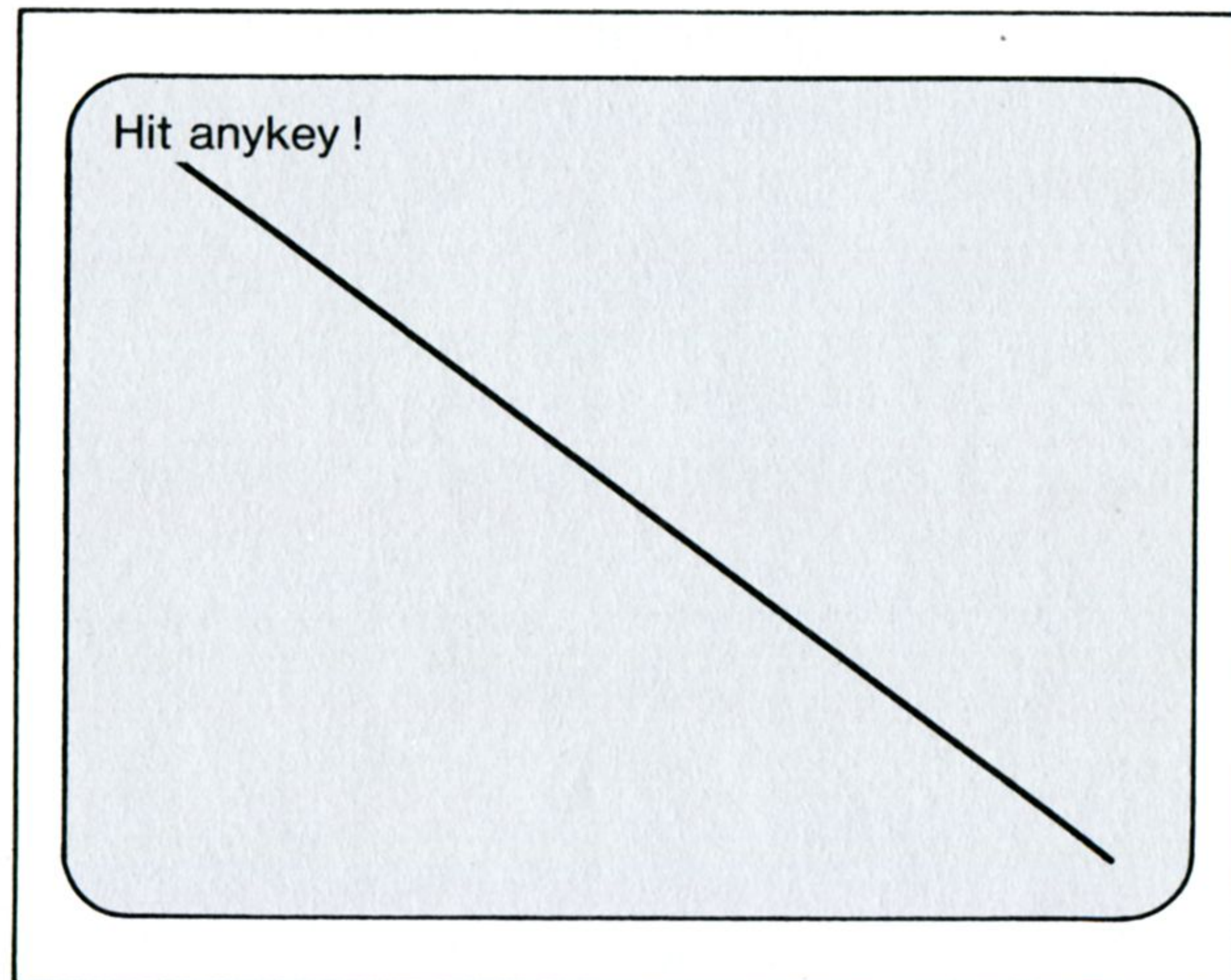
また、[色] は、線を描く色です。ここにはCOLORステートメントで使う番号を指定します。

それでは、実際に図形を描くプログラミングです。

●ソースリストPRG-NO56.BAS

```
CLS
LINE(0, 0)-(639, 399),4'
PRINT "Hit anykey !"
DO
LOOP UNTIL INKEY$ <> ""
```

●インタプリタによる実行結果

```
Hit anykey !
```

画面の左上から右下まで、一直線に赤い線が引けたと思います。

線を描くことが分かったのですから、今度はBOX型の線を引いてみましょう。

BOXを描く

BOX型の線を引くには、LINEステートメントを繰り返せばいいのですが、いちいちLINEステートメントを繰り返すのも面倒です。何かいい方法はないでしょうか。

実は、LINEステートメントにはBOXを描くのに便利なオプションが用意されています。

●●LINEステートメントの書式、BOXを描く●●

LINE　(X1-Y1) - (X2-Y2), [色], B もしくはBF

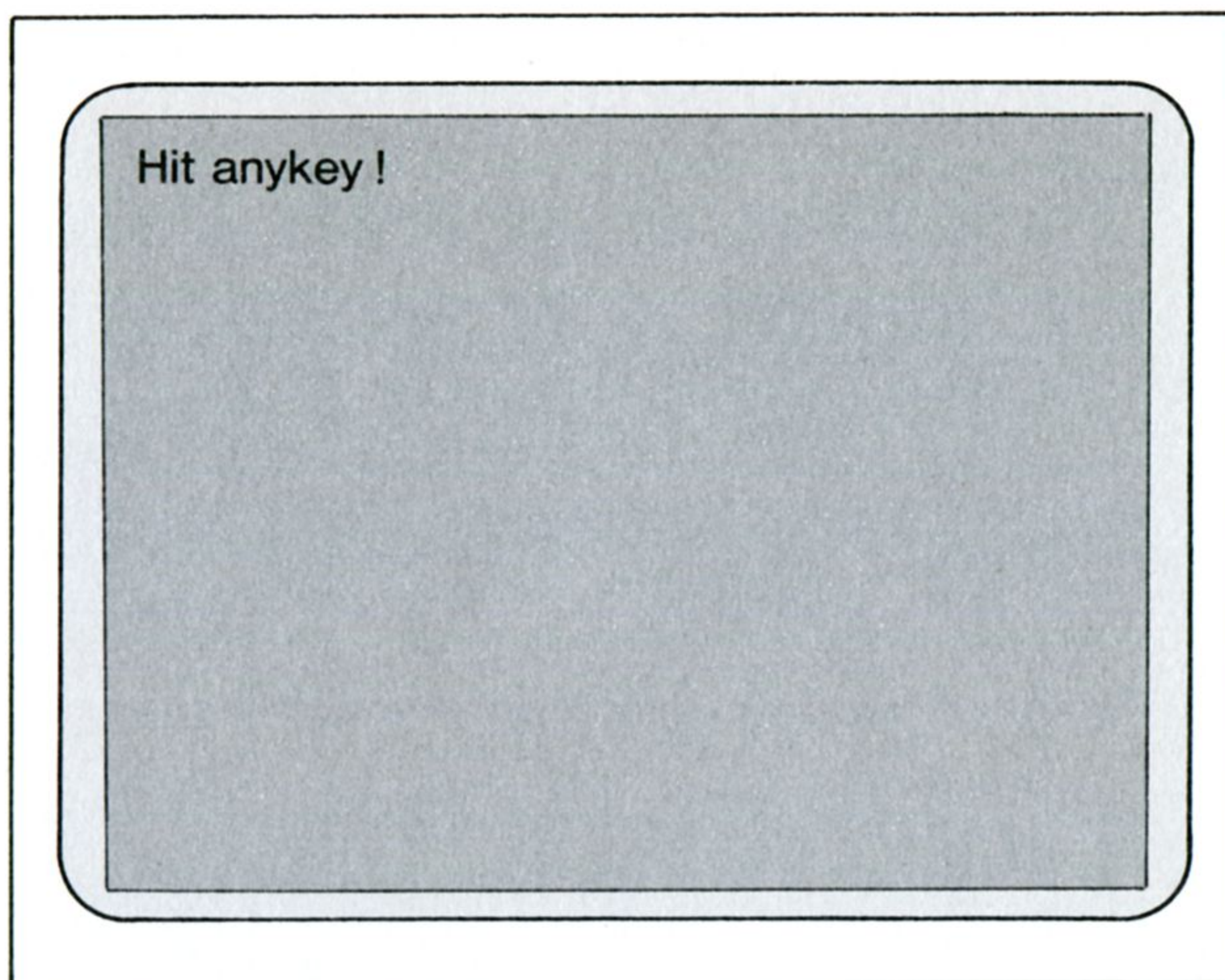
Bオプションを選ぶと、X1・Y1とX2・Y2を対角線とするボックスを描き、BFオプションを選ぶと、X1・Y1とX2・Y2を対角線とするボックスを描いた上、中を指定の色で塗りつぶします。

プログラミングしてみましょう。

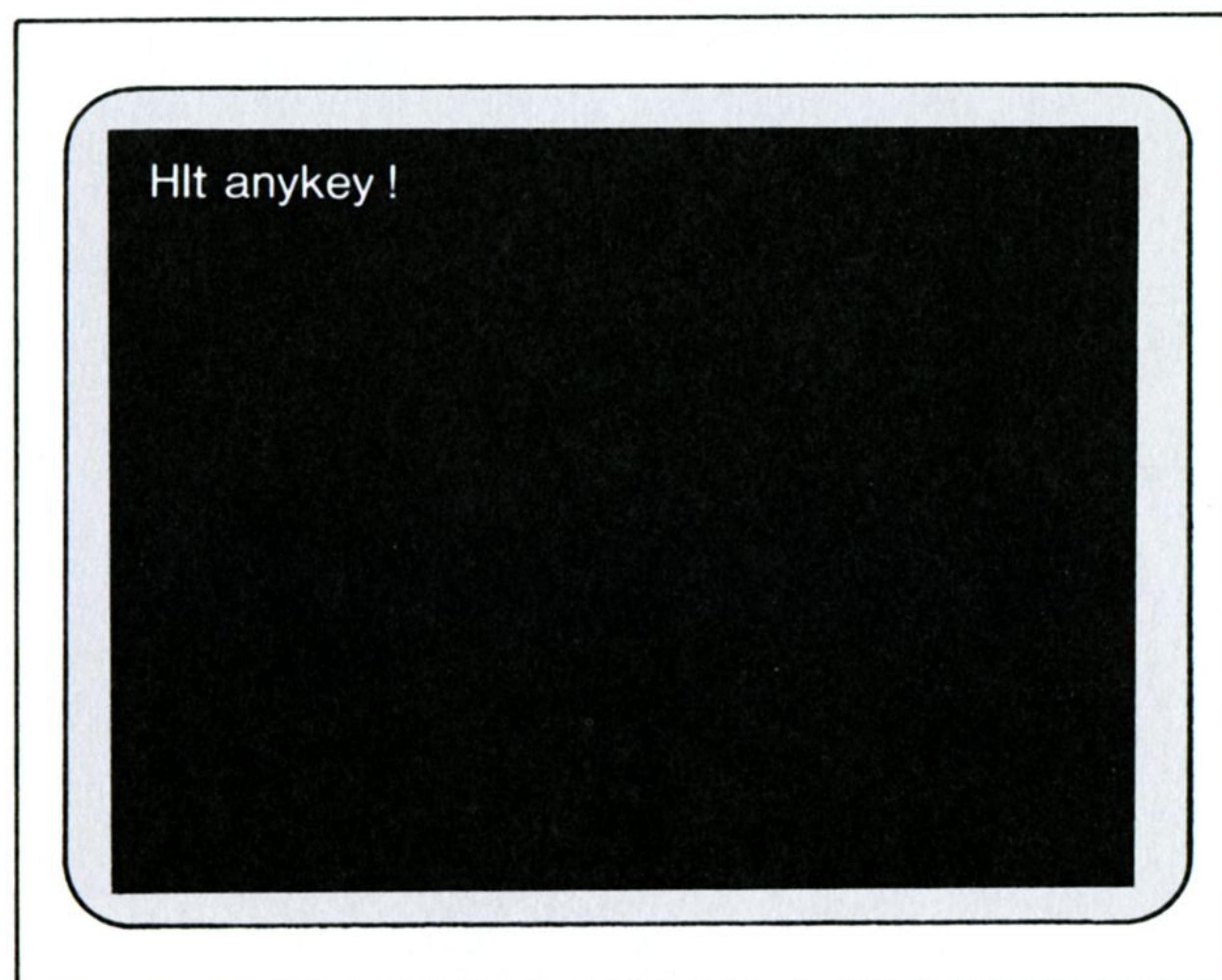
◉ソースリストPRG-NO57.BAS

```
CLS
LINE(0, 0)-(639, 399),4 ,B ←―― これを追加
DO
PRINT "Hit anykey !"
LOOP UNTIL INKEY$ <> ""
```

●インタプリタによる実行結果

Hit anykey !

また、オプションをBFに変更すると次のようになります。

●インタプリタによる実行結果

HIt anykey !

このほかにもBOXを描くテクニックはありますが、LINEステートメントだけで十分だと思います。

グラフィックを描くスピードも早く、また、オプション1つで自由に塗りつぶしたりできるので、便利ですね。

3　アニメーション

さて、ボックスの最後に、ちょっとしたアニメーションを行なってみましょう。

アニメーションの原理は、「絵を描く」、「少しずらして絵を描く」、「元のところにあった絵を背景の絵と同じ色にする」ということの繰り返しです。

この動作を早く行なうことで、あたかも絵が動いているように見えるわけですね。

このような、反復単純作業はコンピュータが得意とするところです。

ここで簡単なアニメーションを作成してみましょう。

◉ソースリストPRG-NO58.BAS

```
CLS
X = 320
Y = 200
PRINT "Hit anykey is start."
LINE (X, Y)-(X + 5, Y + 5), 6, BF      ←移動するブロックを表示
DO
LOOP UNTIL INKEY$ <> ""
CLS
PRINT "Hit anykey is end programs.    X =       Y ="
COLOR 3
                    ブロックの移動点を表示。PRINT△USINGステートメントは，書式付き表示のテクニック
DO
    LOCATE 1, 35: PRINT USING "###"; X
    LOCATE 1, 44: PRINT USING "###"; Y
    LINE (X, Y)-(X + 5, Y + 5), 0, BF
    IF Y < 5 THEN                          表示したブロックを削除。
        FLGY = 0                           バックグランドを同じ色にする
    END IF
    IF Y > 395 THEN
        FLGY = 1
    END IF                                 ①移動するブロックの方向を決定
```

```
IF X < 5 THEN
     FLGX = 0
END IF
IF X > 635 THEN
     FLGX = 1
END IF

IF FLGY = 0 THEN
     Y = Y + 4
END IF                        ←②ブロックの移動
IF FLGY = 1 THEN
     Y = Y - 4
END IF
IF FLGX = 0 THEN
     X = X + 4
END IF
IF FLGX = 1 THEN
     X = X - 4
END IF
                              ブロックの移動量に従いブロックを表示
LINE (X, Y)-(X + 5, Y + 5), 6, BF

FOR Z = 1 TO 5                ←ダミーループ
NEXT                            ブロックをちょっとの間だけ表示

LOOP UNTIL INKEY$ <> ""
COLOR 7
END
```

ブロックが画面の中を動き回るプログラムです。

まず最初に、動かすブロックを表示します。次にブロックを消します。そしてブロックを少しずらして表示します。この繰り返しが本プログラムの主だった作業です。

ただ、ブロックを動かしただけですと、いつかは画面からはみ出てしまうので、画面の範囲を超えたら、ブロックの移動する方向を変更する必要があります。

プログラム上では、①の部分がブロックの移動先を変更するルーチンです。

X･Y共にブロックの移動先を示す値を格納してありますが、これらの値が画面からはみ出る、つまり、0か399もしくは0か639を超えたらブロックの移動先を変更します。

例えば、ブロックが画面左上から右下に向かって移動している場合、X･Yの値はプラスされ続けます。これが、画面の右下にきてしまった場合には、今度はX・Yの値をマイナスすればいいのです。

これをプログラム的に行なっているのが、①ということになります。

また、これらの情報を元に、実際に移動先を決めているのが②です。

ここでは、①で得たブロックの移動方向を元に、X･Yに数値をプラスするのか、もしくはマイナスするのかを実際に行なっています。つまり、実際にX･Yの値から数値をプラスしたりマイナスしたりしています。

これらの値を元に、次の行でブロックを少しずらして表示しています。

次に、ダミーループで時間稼ぎをします。ここで時間を稼ぐことにより、画面に少しの間ブロックが表示されます。

そして、DO...LOOPの先頭に返り、今表示していたブロックを消します。

文章にすると長いですが、実際はかなり高速に行なわれるので、ブロックは付いたり消えたりするのではなく、移動しているように見えることになります。

なお、私は9801 RA2を使っていたので、ダミーループを5回としましたが、実行の遅いマシンの場合には、ダミーループを削ってしまっても構いません。

また、それでも満足のいかない速度でブロックが動く場合には、画面にブロックの現在位置を表示するPRINT USING...の2行を削除してください。

```
LOCATE 1, 35: PRINT USING "###"; X
LOCATE 1, 44: PRINT USING "###"; Y
```

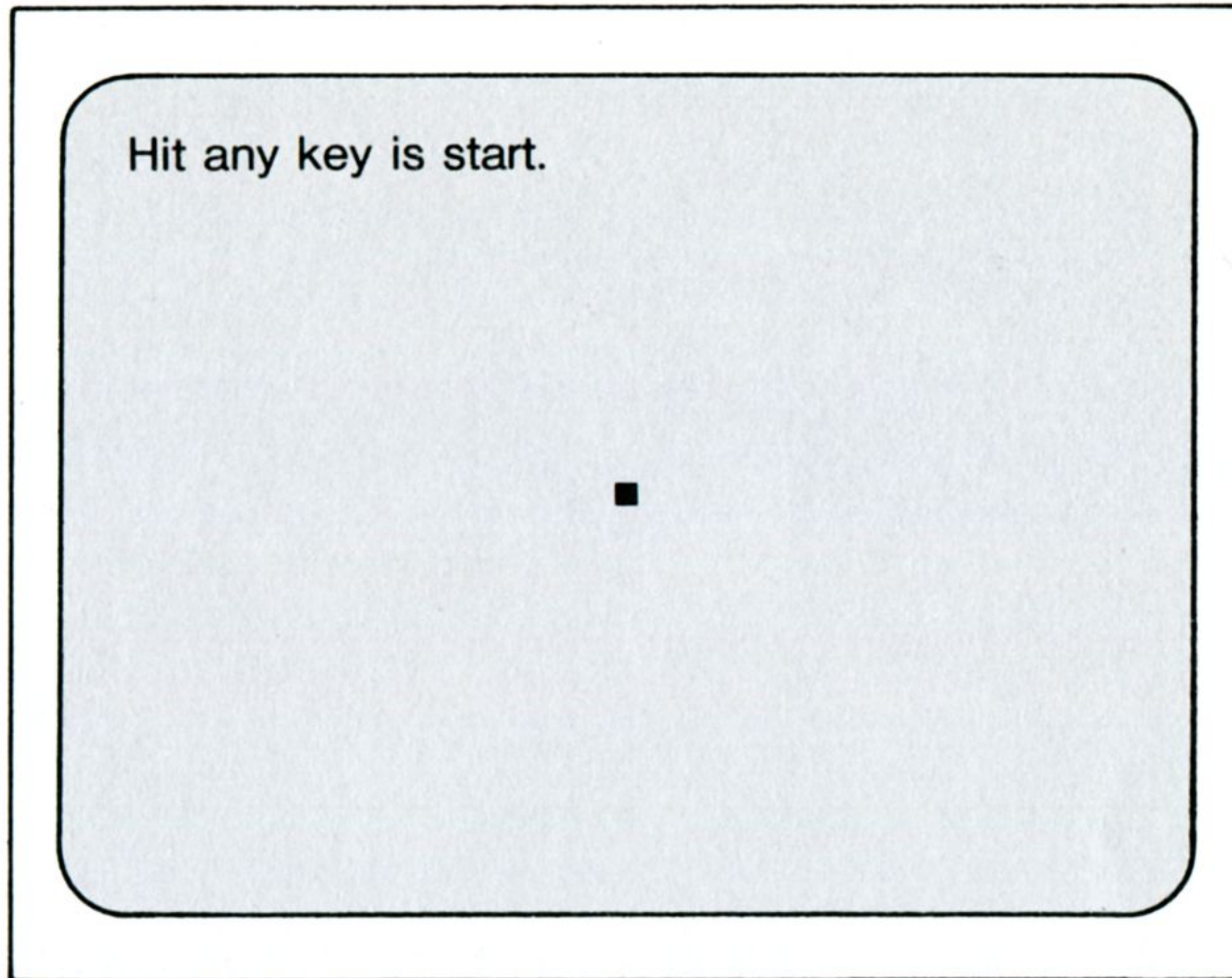
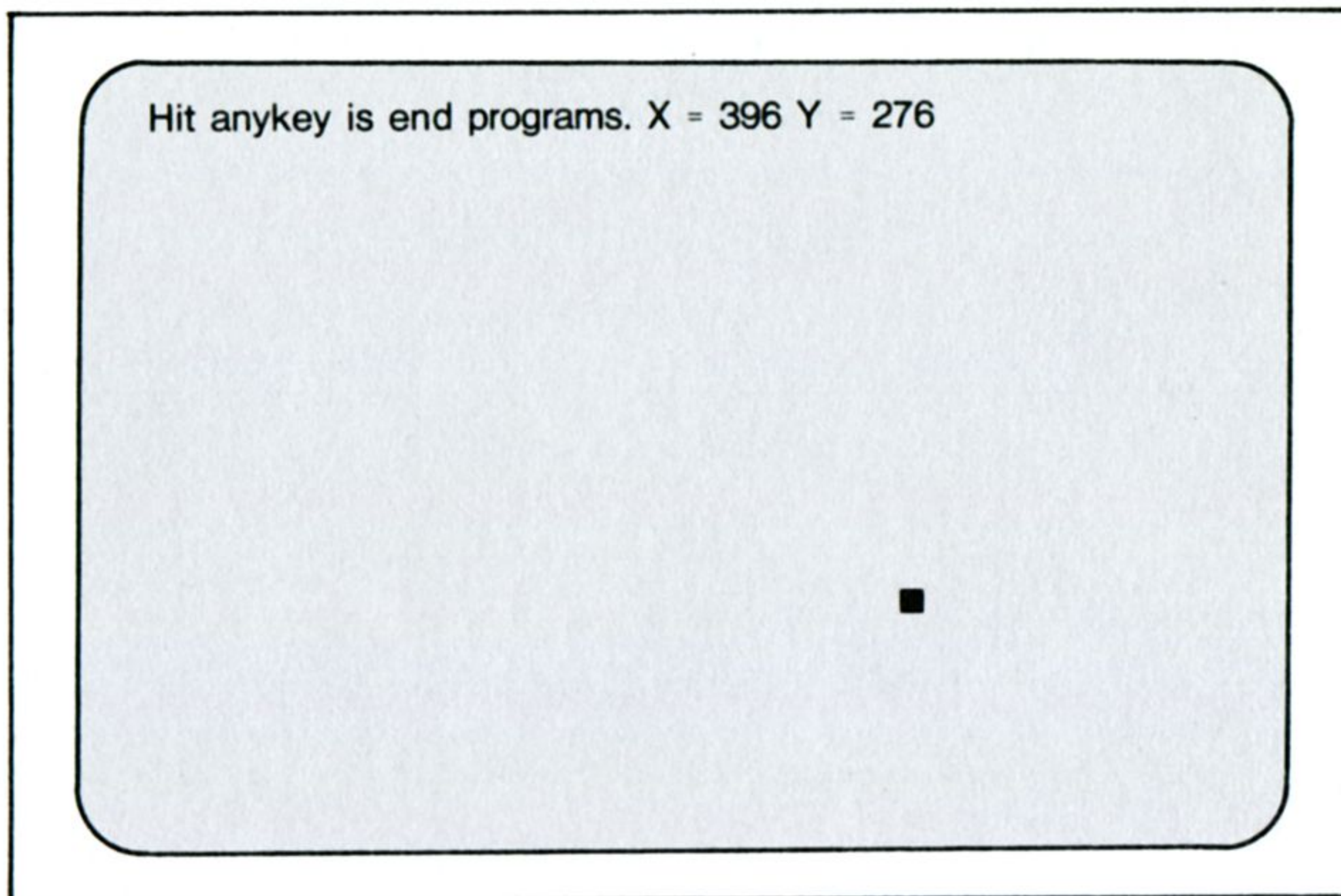
実行してみましょう。

●インタプリタによる実行結果

①初期画面

```
Hit any key is start.
```

②ブロックが動いているところ

```
Hit anykey is end programs. X = 396 Y = 276
```

キーボードコード表

Quick BASICでは、ASCIIコードと呼ばれる、統一企画のコードとスキャンコードと呼ばれる、PC98シリーズの内部コードの直接指定とがあります。ここでは、これら両方のコードを示します。

なお、コードはすべて16進数で表記してあります。

キー	スキャンコード	ASCII	SHIFT	CTRL	GRPH
ESC	00	1B	1B	1B	
1　!	01	31	21		
2　”	02	32	22		
3　#	03	33	23		
4　$	04	34	24		
5　%	05	35	25		
6　&	06	36	26		
7　’	07	37	27		
8　(	08	38	28		
9　)	09	39	29		
0	0A	30	30		
−　=	0B	2D	3D		
^　^	0C	5E	5E	1E	
¥　|	0D	5C	7C	1C	
BS	0E	08	08	08	
TAB	0F	09	09	09	
Q	10	71	51	11	
W	11	77	57	17	
E	12	65	45	05	
R	13	72	52	12	

キー	スキャンコード	ASCII	SHIFT	CTRL	GRPH
T	14	74	54	14	
Y	15	79	59	19	
U	16	75	55	15	
I	17	69	49	09	
O	18	6F	4F	0F	
P	19	70	50	10	
@ ～	1A	40	7E		
[{	1B	5B	7B	1B	
RETURN	1C	0D	0D	0D	
CTRL	74				
CAPS	71				
A	1D	61	41	01	
S	1E	73	53	13	
D	1F	64	44	04	
F	20	66	46	06	
G	21	67	47	07	
H	22	68	48	08	
J	23	6A	4A	0C	
K	24	6B	4B		
L	25	6C	4C		
; +	26	3B	2B		
: *	27	3A	2A		
] }	28	5D	7D	1D	
L SHIFT	70				
Z	29	7A	5A	1A	
X	2A	78	58	18	
C	2B	63	43	03	

キー	スキャンコード	ASCII	SHIFT	CTRL	GRPH
V	2C	76	56	16	
B	2D	62	42	02	
N	2E	6E	4E	0E	
M	2F	6D	4D	0D	
, <	30	2C	3C		
. >	31	2E	3E		
／ ?	32	2F	3F		
＿	33		5F	1F	
R SHIFT	70	カナ	72		
GRPH	73				
NFER	51				
SPACE	34	20	20	20	
XFER	35	STOP	60		
COPY	61				
f･1	62	00＋3B	00＋54		
f･2	63	00＋3C	00＋55		
f･3	64	00＋3D	00＋56		
f･4	65	00＋3E	00＋57		
f･5	66	00＋3F	00＋58		
f･6	67	00＋40	00＋59		
f･7	68	00＋41	00＋5A		
f･8	69	00＋42	00＋5B		
f･9	6A	00＋43	00＋5C		
f･10	6B	00＋44	00＋5D		
ROLL UP	36	00＋51	00＋51		
ROLL DOWN	37	00＋49	00＋49		

キー	スキャンコード	ASCII	SHIFT	CTRL	GRPH
INS	38	00＋52	00＋52		
DEL	39	00＋53	00＋53		
↑	3A	00＋48	00＋48		
←	3B	00＋4B	00＋4B		
→	3C	00＋4D	00＋4D		
↓	3D	00＋50	00＋50		
HOME	3E	00＋47	00＋47		
HELP	3F	00＋4F	00＋4F		
−	40	40	40		
／	41	41	41		
7	42	42	42		
8	43	43	43		
9	44	44	44		
＊	45	45	45		
4	46	46	46		
5	47	47	47		
6	48	48	48		
＋	49	49	49		
1	4A	4A	4A		
2	4B	4B	4B		
3	4C	4C	4C		
＝	4D	4D	4D		
0	4E	4E	4E		
,	4F	4F	4F		
.	50	50	50		
RERURN	1C	1C	1C		

索　引

◉記号

◉アルファベット

◉あ行

◉か行

◉さ行

◉た行

◉は行

◉ま行

◉ら行

◉わ行

はじめて学ぶ

Quick BASIC

1989年6月10日　初版発行

著　　者	織田純一
編　　集	株式会社新紀元社編集部
発 行 者	米倉文吉
発 行 所	株式会社新紀元社
	東京都千代田区神田錦町2-9　麻生ビル〒101
	電話　03-291-0961
	FAX　03-291-0963
	郵便為替　東京1-27618
表紙意匠	渡辺葉子
製　　版	新潮製版株式会社
印刷・製本	株式会社文化印刷・株式会社エイエヌオフセット

ISBN4-915146-08-1

定価はカバーに表示してあります。